

Inteligencia Artificial

Manual de Prácticas



Verónica Esther Arriola Ríos
Rodrigo Eduardo Colín Rivera
Luis Alfredo Lizárraga Santos
Benjamín Torres Saavedra
Víctor Germán Mijangos de la Cruz

Edición

Agradecemos especialmente a:

Luis Alfredo Lizárraga Santos

Por haber realizado la primer compilación y edición de este manual, sucesivas actualizaciones se han realizado sobre su trabajo.

Índice general

Índice general	I
I Prácticas	1
1 Introducción a Agentes	3
1.1 Objetivo	3
1.2 Introducción	3
1.2.1 Modelo basado en agentes	4
1.2.2 Agentes autónomos y auto-organización	4
1.2.3 Modelo de termitas	5
1.3 Desarrollo e implementación	5
1.3.1 Implementación	6
1.4 Requisitos y resultados	8
2 Estados y espacio de búsqueda	11
2.1 Objetivo	11
2.2 Introducción	11
2.3 Desarrollo e implementación	13
2.3.1 Implementación	14
2.3.2 Punto extra	15
2.4 Requisitos y resultados	15
3 Algoritmos Genéticos	16
3.1 Objetivo	16
3.2 Introducción	16
3.2.1 Metodología	16
3.2.2 Requisitos del algoritmo genético	17
3.2.3 Etapas del algoritmo genético	17
3.3 Desarrollo e implementación	19
3.3.1 Consideraciones de la implementación	20
3.3.2 Pseudocódigo	22
3.4 Requisitos y resultados	22
4 Retractación	24

ÍNDICE GENERAL

4.1	Objetivo	24
4.2	Introducción	24
4.3	Desarrollo e implementación	25
4.3.1	Algoritmo de construcción del laberinto	25
4.3.2	Implementación	27
4.4	Requisitos y resultados	27
5	Recocido simulado	28
5.1	Objetivo	28
5.2	Introducción	28
5.2.1	Recocido simulado	28
5.2.2	Problema del agente viajero (TSP)	28
5.3	Desarrollo e implementación	29
5.3.1	Implementación	30
5.3.2	Punto extra	31
5.4	Requisitos y resultados	32
6	A* Pakuman	33
6.1	Objetivo	33
6.2	Introducción	33
6.2.1	Navegación	34
6.2.2	Iniciando la búsqueda	35
6.2.3	Puntuando el camino	36
6.2.4	Continuando la búsqueda	38
6.3	Desarrollo e implementación	38
6.3.1	Implementación	40
6.4	Requisitos y resultados	43
7	Perceptrón	45
7.1	Objetivo	45
7.2	Introducción	45
7.2.1	Redes Neuronales Artificiales	45
7.2.2	Neuronas Artificiales	47
7.2.3	Perceptrón	47
7.2.4	Resumen del algoritmo de aprendizaje	48
7.3	Desarrollo e implementación	49
7.4	Requisitos y resultados	50
8	Regresión	51
8.1	Objetivo	51
8.2	Introducción	51
8.2.1	Regresión logística lineal	52
8.2.2	Estimación del modelo	54
8.2.3	Clasificación con regresión logística	55

8.3 Desarrollo	56
8.3.1 Implementación	56
8.3.2 Requisitos y resultados	57
9 Factores	58
9.1 Objetivo	58
9.2 Introducción	58
9.2.1 Operaciones	59
9.3 Desarrollo e implementación	63
9.4 Requisitos y resultados	65
10 Inferencia	67
10.1 Objetivo	67
10.2 Introducción	67
10.3 Desarrollo e implementación	68
10.4 Requisitos y resultados	68
11 Bayes Ingenuo	71
11.1 Objetivo	71
11.2 Introducción	71
11.2.1 Clasificador Bayesiano Ingenuo	71
11.2.2 Utilizando un Clasificador Bayesiano	72
11.2.3 Propiedades	72
11.2.4 Teorema de Bayes	73
11.2.5 Clasificador ingenuo	73
11.2.6 Calculando probabilidades	74
11.2.7 Mejorando el clasificador	75
11.3 Desarrollo e implementación	77
11.4 Requisitos y resultados	79
12 Cadenas de Márkov	80
12.1 Objetivo	80
12.2 Introducción	80
12.2.1 Inferencia	81
12.2.2 Ejemplo: El soldado	82
12.2.3 Estado estacionario	83
12.2.4 Ejemplo: comportamiento del soldado en el estado estacionario	84
12.3 Desarrollo	84
12.4 Requisitos y resultados	86
13 Viterbi	87
13.1 Objetivo	87
13.2 Introducción	87
13.2.1 Estimación del modelo	89
13.2.2 Etiquetado con HMMs	90

ÍNDICE GENERAL

13.2.3 Algoritmo de Viterbi	91
13.2.4 Ejemplo: etiquetado del lenguaje natural	94
13.3 Desarrollo	96
13.3.1 Implementación	96
13.4 Requisitos y resultados	98
II Proyectos	99
14 STRIPS	100
14.1 Meta	100
14.2 Objetivos	100
14.3 Desarrollo	100
14.3.1 Escenario de prueba	101
14.3.2 Tareas a realizar	103
15 Lego Mindstorms	104
15.1 Objetivo	104
15.2 Introducción	104
15.2.1 ¿Que es Lego Mindstorms?	104
15.2.2 Instalando ev3dev	104
15.2.3 Conectándose a internet	106
15.2.4 Instalando Python, virtualenv y bibliotecas extras	106
15.3 Desarrollo e implementación	107
15.3.1 Robot	107
15.3.2 Algoritmo	107
15.4 Requisitos y resultados	109
16 Localización de robots	110
16.1 Objetivo	110
16.2 Introducción	110
16.2.1 Localización Robótica	110
16.2.2 Localización de Markov	111
16.3 Desarrollo	113
16.3.1 Simulación del ambiente y del robot	113
16.3.2 Implementación del modelo de Markov	115
16.3.3 Notación	117
16.3.4 Algoritmo	118
16.4 Desarrollo e implementación	119
16.4.1 Tips	119
16.5 Requisitos y resultados	119

III Apéndices	120
A Código de prácticas	121
A.1 Lego Mindstorms	121
Bibliografía	123

PARTE I

Prácticas

Convenciones

A lo largo del texto se utilizará la siguiente notación para diversos elementos:

Conjuntos	C
Vectores	x
Matrices	M
Unidades	cm

1 | Introducción a Agentes

Rodrigo Eduardo Colín Rivera

Objetivo

Que el alumno se familiarice con la abstracción del concepto de agente mediante la programación de un sistema de simulación biológico que muestra las bases de un autómata celular, programación dirigida a agentes, sistemas complejos y emergencia de propiedades como la auto-organización.

Introducción

Un autómata celular es un modelo discreto que consiste en una cuadrícula de células, cada una con un número finito de estados. La cuadrícula puede estar en cualquier número finito de dimensiones pero lo más común es encontrar autómatas en una, dos y tres dimensiones para que tengan sentido geométrico. Cada célula tiene definido un conjunto de células llamado vecindad [Figura 1.1 y Figura 1.2].

Se tiene un estado inicial (al tiempo $t=0$) en el que se asigna un estado a cada célula. Una nueva generación es creada (avanzar t en 1) según alguna regla que determina el nuevo estado de cada célula en términos del estado actual de la célula y la de sus vecinos

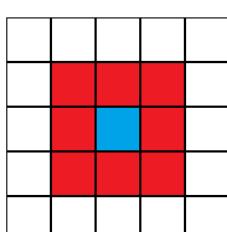


Figura 1.1 Vecindad de Moore

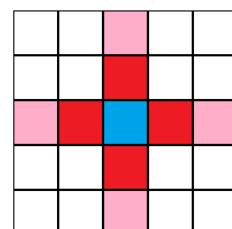


Figura 1.2 Vecindad de Von Neumann de radio 1 (rojo) y radio 2 (rojo y rosa).

1. Introducción a Agentes

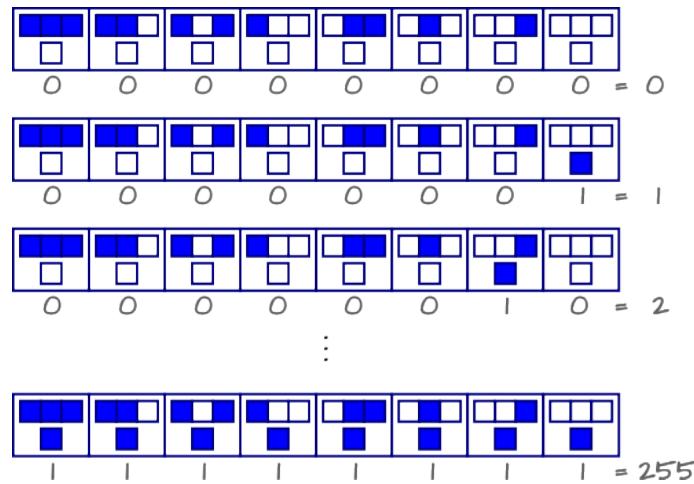


Figura 1.3 Numeración de reglas de transición para autómatas celulares unidimensionales.

[Figura 1.3].

Modelo basado en agentes

Un modelo basado en agentes es un tipo de modelo computacional que permite la simulación de acciones e interacciones entre individuos autónomos dentro de un entorno, y permite determinar qué efectos producen en el conjunto del sistema.

Combina elementos de teoría de juegos, sistemas complejos, emergencia, sociología computacional, sistemas multi-agente y programación evolutiva. Los modelos simulan las operaciones simultáneas e interacciones de múltiples agentes, en un intento de recrear y predecir la apariencia de fenómenos complejos. El proceso de emergencia surge de un nivel bajo hacia niveles del sistema más altos. La clave es notar que reglas de comportamientos sencillos generan comportamientos complejos.

Agentes autónomos y auto-organización

Un agente autónomo es una unidad que interactúa con su entorno (el cual probablemente consta de otros agentes) pero actúa independientemente de todos los demás agentes porque no toma decisiones con respecto a algún líder o plan global a seguir. Es decir, cada agente actúa por sí mismo.

Así, veremos cómo múltiples agentes pueden desempeñar tareas que aparentan seguir un plan global. A este proceso en el que cada agente autónomo interactúa a su propia manera para crear un orden global se le conoce como auto-organización y se observa cómo modelos simples son capaces de generar comportamientos complejos.

Modelo de termitas

Mitchel Resnick 1994 estudió varios sistemas de agentes primitivos, uno de ellos fueron las termitas teóricas dentro de un espacio con astillas esparcidas que seguían las siguientes reglas:

- Caminar aleatoriamente hasta encontrar una astilla.
- Si la termita está cargando una astilla, la suelta y continua caminando aleatoriamente.
- Si la termita no está cargando una astilla, la toma y continua caminando aleatoriamente con la astilla.

Claramente, las reglas definidas por Resnick son tan simples como es posible. No parece haber lugar para un comportamiento inteligente en un modelo como éste, tampoco parece que las termitas puedan producir nada con algún sentido más allá de la aleatoriedad de las astillas distribuidas en el entorno.

La Figura 1.4 muestra seis escenarios de la simulación del conjunto de reglas simples con un conjunto pequeño de termitas. En la configuración inicial el universo de termitas consiste en una cuadrícula con astillas aleatoriamente distribuidas. La representación de la cuadrícula consta de una frontera periódica, es decir, un punto en una de las aristas de la cuadrícula tiene como vecinos a los puntos en la arista opuesta. Al comenzar la simulación, las termitas mueven las astillas en pequeños grupos o clusters. Conforme pasa el tiempo, los clusters se vuelven más grandes y más definidos.

Tras cientos de miles de pasos en la ejecución de la simulación, como se muestra en la última imagen de la Figura 1.4, las astillas están claramente bien definidas en una colección. Obviamente esto es un método poco óptimo para coleccionar astillas, sin mencionar lo frustrante que es observar el proceso. Sin embargo, con el paso del tiempo es un hecho que el orden del sistema es evidente como resultado.

Desarrollo e implementación

Los resultados anteriores son reportados por el propio Resnick, sin embargo, para mostrar que es posible llegar al mismo resultado, se lleva a cabo una implementación con el lenguaje de programación Processing. Al alumno se le proporciona parte de la implementación, de manera que se enfoque únicamente en programar el comportamiento descrito y evitando perder tiempo en la interfaz gráfica.

Las clases, métodos y variables más relevantes son las siguientes:

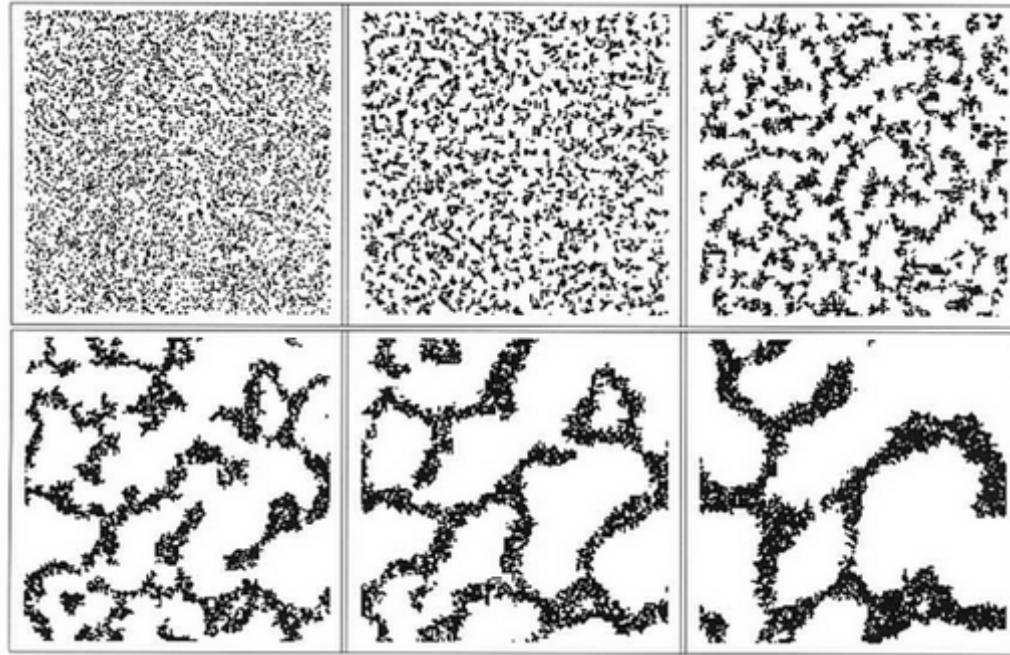


Figura 1.4 Termitas colocando aleatoriamente astillas con las reglas definidas anteriormente, tomada de Resnick 1994.

Clase Celda

Representación de cada espacio dentro de la cuadrícula donde estarán las termitas y astillas. Cada celda tiene coordenadas (x,y) y un valor booleano para indicar si hay una astilla.

Clase Termita

Representación de una termita. Se representan las coordenadas (x,y) de su posición, la dirección en la que está observando (auxiliar que más adelante será mencionado a detalle) y un valor booleano para indicar si está cargando una astilla.

Clase ModeloTermitas

Representación de una colonia de termitas. Principalmente contiene una matriz de celdas (la representación del mundo) y una lista de termitas (nuestros agentes). Adicionalmente se define la cantidad de celdas a lo ancho y alto, un valor auxiliar para conocer la cantidad de iteraciones, un objeto de la clase Random (para hacer decisiones aleatorias) y el tamaño en pixeles de cada celda (para la visualización con Processing).

Implementación

El constructor de la clase `ModeloTermitas` ya está implementado (principalmente para inicializar el espacio y termitas). También se encuentra implementado el método

0	1	2
7		3
6	5	4

Figura 1.5 Las 8 posibles direcciones y vecindades de cada termita o celda.

moverTermita, que mueve la termita dada como parámetro en la dirección indicada. Cada termita tiene una vecindad de Moore, es decir, tienen 8 celdas adyacentes (considerando un espacio periódico) que se enumeran según la Figura 1.5.

De la misma manera se indica la dirección en la que puede mirar una termita. Por ejemplo, si la termita está mirando en dirección con valor 1 significa que está observando hacia arriba. Si tuviera el valor 4 significa que está observando en diagonal inferior derecha.

Existen 3 maneras diferentes de simular e implementar el modelo de termitas:

1. Usando las 8 posibles direcciones

Siguiendo la idea original con caminatas aleatorias en las 8 posibles direcciones para las termitas.

2. Modificar la caminata para que sea aleatoria y restringida

Una manera de avanzar totalmente aleatoria como en el primer caso implica que pueden existir muchos movimientos innecesarios (considerese la situación en la que una termita se cuela moviéndose en la casilla delante de ella y detrás de ella). Empleando la dirección en la que está observando la termita se puede restringir su movimiento a solo 3 opciones: a la izquierda, al frente o a la derecha. Adicionalmente al soltar una astilla, la termita da media vuelta y se coloca en la dirección opuesta de donde soltó la astilla (esto evita una situación similar en la que se cicle una termita moviendo la misma astilla al mismo lugar).

3. Brindar un salto a las termitas

Esto significa que en el momento en que una termita suelta una astilla, en lugar de moverse en la dirección opuesta, las termitas “brincan” o se mueven a una celda sin astilla y continúan caminando de manera aleatoria y restringida.

Cada una de las diferentes maneras de implementación se encuentran asignadas a los métodos evolucion1, evolucion2 y evolucion3, respectivamente.

El archivo termitas/Termitas.java contiene parte del código de la simulación y solamente tiene implementada la visualización de las termitas como cuadrados verdes moviéndose aleatoriamente [Figura 1.6]. Al implementar todos los métodos faltantes se darán cuenta de que cuando una termita está cargando una astilla (cuadros amarillos) cambia de color a rojo.

1. Introducción a Agentes

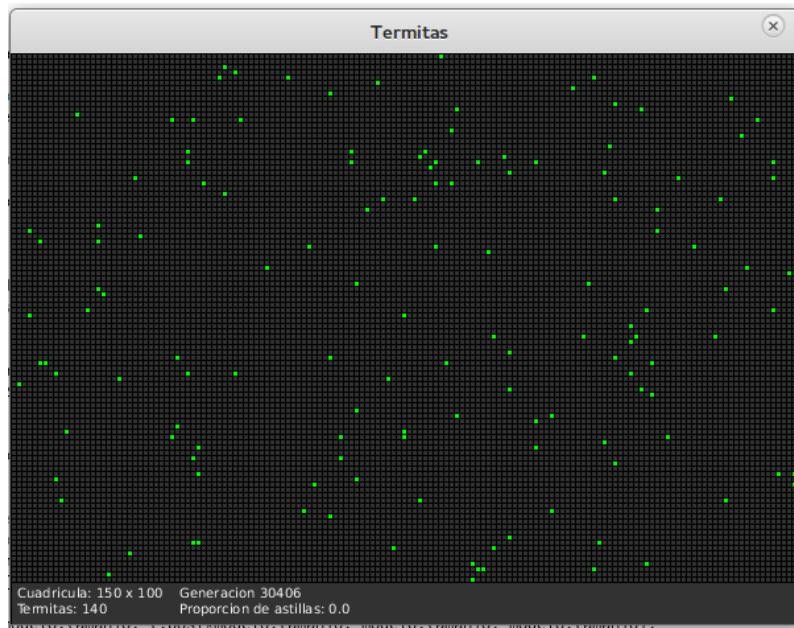


Figura 1.6 Captura de pantalla de `Termitas.java`, que hace uso de Processing.

Se debe implementar el comportamiento de las termitas para simular todo el sistema de la mejor manera. Dado que la interfaz gráfica está dada, solamente es necesario implementar los siguientes métodos:

- `int direccionAleatoriaFrente(int direccion)`
- `boolean hayAstilla(Termita t, int direccion)`
- `void dejarAstilla(Termita t, int direccion)`
- `void dejarAstilla(Termita t)`
- `void dejarAstillaConSalto(Termita t)`
- `void tomarAstilla(Termita t, int direccion)`

Cada método se encuentra especificado dentro del archivo `Termitas.java`.

Requisitos y resultados

Para evaluar y calificar la práctica es necesario que se implementen todos los métodos mencionados e indicados en el código, respetando implementar sólo lo que se pide (para evitar comportamientos extraños de la simulación). Es completamente válido utilizar

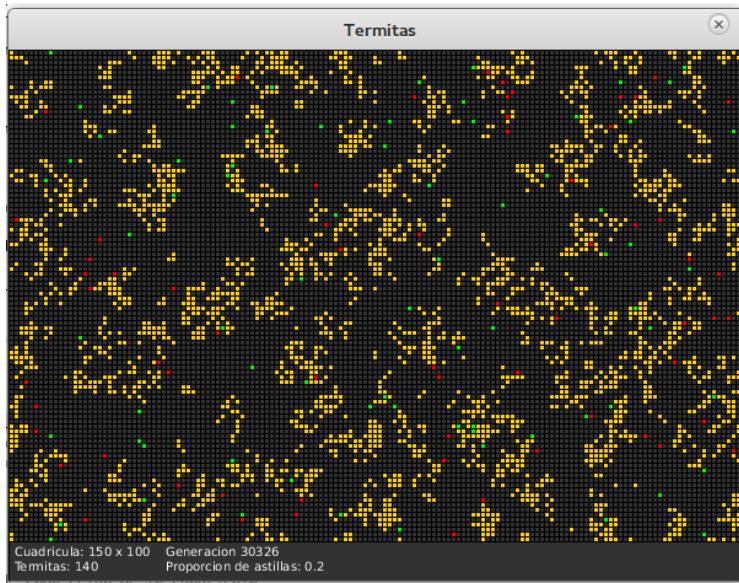


Figura 1.7 Simulación con idea original después de 10000 iteraciones (evolucion1).

bibliotecas adicionales si lo consideran necesario, así como la creación y uso de sus propios métodos auxiliares si lo desean.

Debe notarse una mejora significativa mediante la implementación de la caminata restringida o el uso del salto. Es decir, verifiquen que tras varias iteraciones su implementación del modelo actúe como se espera. Las siguientes imágenes ilustran parte de los resultados esperados [Figuras 1.7, 1.8 y 1.9].

1. Introducción a Agentes

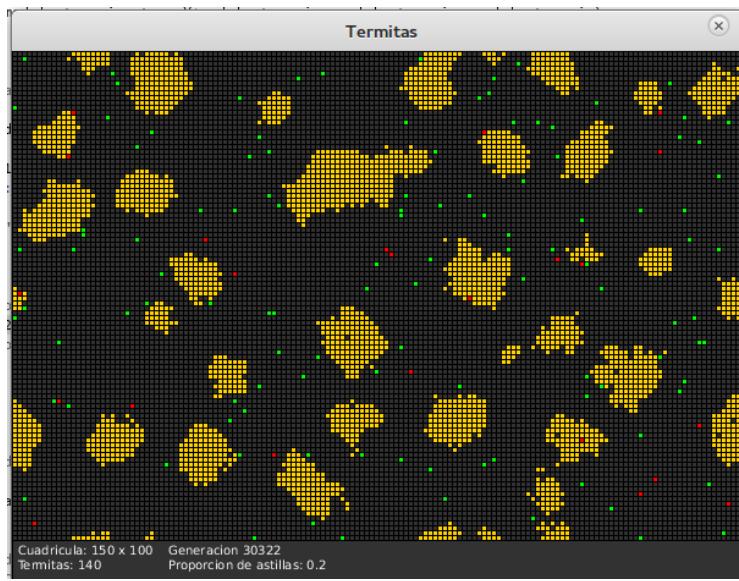


Figura 1.8 Empleando caminata aleatoria restringida tras 5000 iteraciones. La cantidad de astillas es la misma pero se observa un mejor ordenamiento y en menor tiempo.

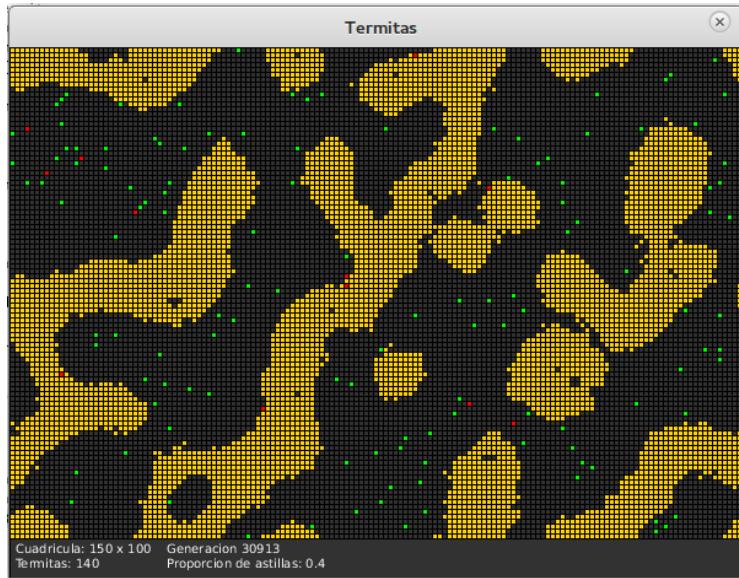


Figura 1.9 El uso de termitas con salto brinda una aproximación similar a la anterior. Cambiando algunos parámetros se pueden obtener resultados similares a colonias de termitas de la naturaleza.

2 | Estados y espacio de búsqueda

Rodrigo Eduardo Colín Rivera
Verónica Esther Arriola Ríos

Objetivo

Que el alumno comprenda la noción de *Estado* y *Espacio de Búsqueda*, y que pueda representar en una estructura de datos todos los posibles estados de un mundo dado.

Introducción

Dado un problema o modelo es posible determinar las características del mundo descrito en él. Uno de los puntos a tratar es la representación de dicho mundo y el concepto de estado. Un estado es una configuración posible del mundo con el que se trabaja. Por ejemplo, un juego de ajedrez se compone de un tablero de 64 cuadros donde inicialmente están 32 piezas (16 para cada jugador). El estado inicial del ajedrez se muestra en la Figura 2.1.

Cuando una pieza se mueve, el estado del ajedrez cambia; pues la posición y cantidad de las piezas en juego determinan el estado del ajedrez. De esta manera el *espacio de estados* se representa como una gráfica donde cada nodo representa un estado del problema y las aristas que los unen son la aplicación de una *acción* (u *operador aterrizado*).

Es decir, el espacio de estados son todas las posibles configuraciones del problema. En el caso del ajedrez, las acciones que unen cada estado en la gráfica representan el movimiento de una pieza por parte del jugador y son el otro elemento requerido para el definir el *sistema de transiciones de estados*.Figura 2.2

¹Milton A. Ramírez Klapp, Notas de Inteligencia Artificial, Universidad San Sebastián, Facultad de Ingeniería y Tecnología.

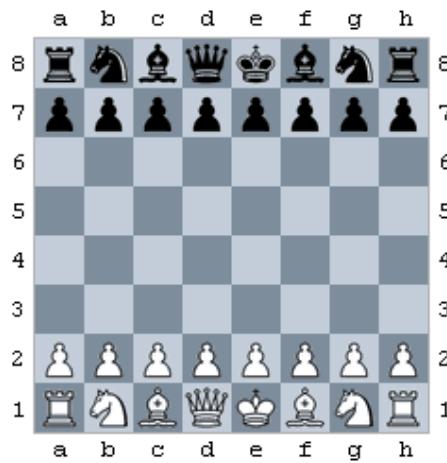


Figura 2.1 Estado inicial del juego de ajedrez.

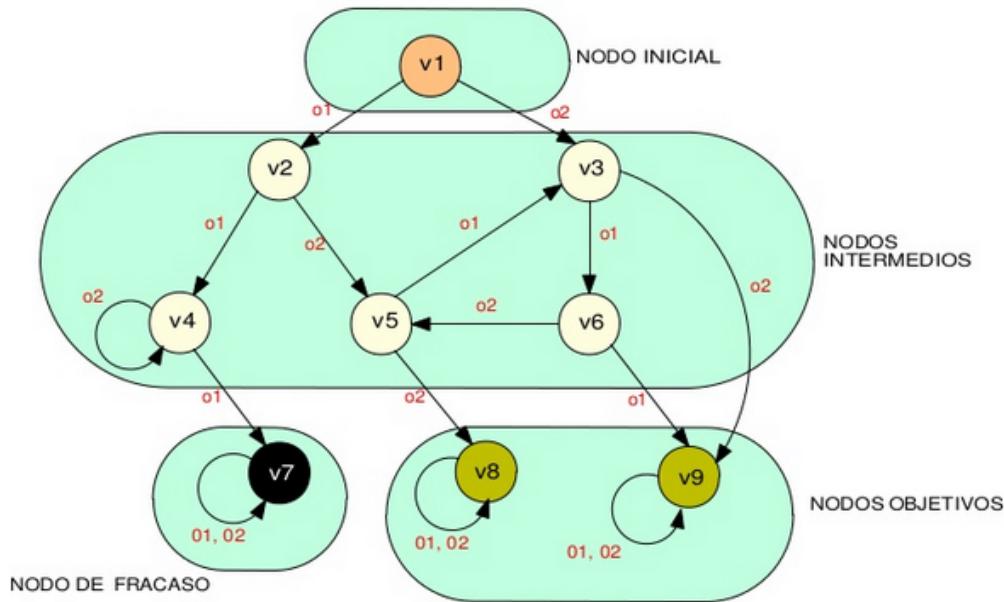


Figura 2.2 Ejemplo de un espacio de estados con las transiciones entre ellos.¹

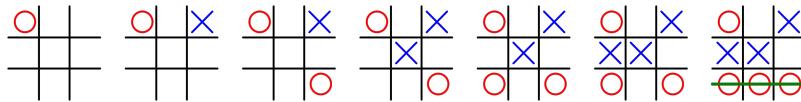
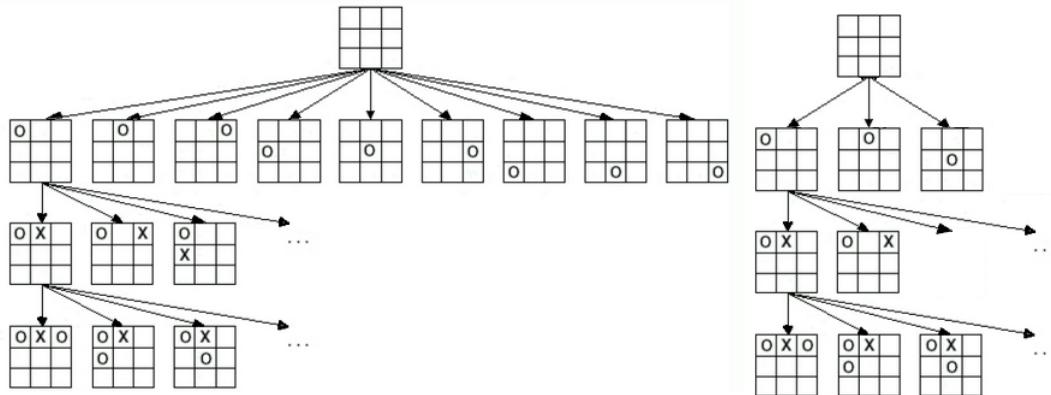


Figura 2.3 Ejemplo de un juego de Gato donde gana el jugador «O».



Izquierda: Espacio de búsqueda en el juego del Gato representado como un árbol sin hacer detección de simetrías. **Derecha:** Espacio de búsqueda en el juego del Gato representado como un árbol eliminando estados equivalentes por simetría.

Desarrollo e implementación

La práctica consiste en generar el sistema de transiciones de estados para el juego del Gato (también llamado *Tres en línea*). Para ello se requiere calcular a los sucesores de cada estado válido del juego y ensamblar el grafo de transiciones. Para generar y almacenar este grafo se utiliza una estructura conocida como *nodo de búsqueda*. Para esta práctica el tipo nodo corresponderá a la clase Gato, la cual se caracteriza por contener la información del estado que contiene y la lista de nodos con los estados sucesores alcanzables mediante alguna jugada desde el nodo actual.

Recordemos que el juego del Gato es entre dos jugadores, representados por los símbolos «O» y «X», que toman turnos para marcar los espacios vacíos de un tablero de 3x3. Un jugador gana cuando logra tener una línea, ya sea horizontal, vertical o diagonal, de 3 símbolos correspondientes. Figura 2.3

Afortunadamente el juego del Gato es lo bastante simple como para evitar ciclarse al generar el espacio de estados y eso es una característica importante que ayuda a la construcción del espacio de estados, ya que podemos hacer uso de un árbol (que es un tipo especial de gráfica) para almacenar los estados.

Por tanto, se necesitan los siguientes elementos para representar el espacio de estados del Gato:

1. Representación del estado

2. Estados y espacio de búsqueda

Una manera de representar el estado del Gato es haciendo uso de una matriz 3×3 :

```
int[][] tablero = {{1,0,1}, {0,0,0}, {4,4,0}}
```

	0	1	2
0	O		O
1			
2	X	X	

Figura 2.5 Representación gráfica del estado del Gato

Con 1 representando al primer jugador y 4 al segundo (fig 2.5).

2. Función generadora

Se necesitará implementar una función que genere los sucesores de un estado del juego del Gato, considerando sólo las jugadas válidas y descartando simetrías. Esto es: si dos tableros son iguales bajo rotación ($90^\circ, 180^\circ, 270^\circ$) o reflexión (horizontal, vertical y diagonal) basta con generar uno de ellos, pues las jugadas que le sigan seguirán siendo idénticas bajo la misma operación de simetría. Figura 2.4

3. Función de comprobación

Se necesitará implementar una función que verifique si hubo ganador en un estado dado, utilizando una bandera para indicar que no se debe generar sucesores de ese estado, para evitar generar estados inalcanzables.

Implementación

Se volverá a trabajar con Processing. Se debe programar lo referente a la generación de sucesores de un estado, verificación de simetrías y agregar variables que lleven el conteo de empates y juegos ganados por cada jugador, esto se imprimirá junto con la información de los nodos del nivel generado.

En pocas palabras es necesario implementar los siguientes métodos:

- `LinkedList<Gato> generaSucesores()`
- `boolean esSimétricoDiagonalInvertida(Gato otro)`
- `boolean esSimétricoDiagonal(Gato otro)`
- `boolean esSimétricoVerticalmente(Gato otro)`
- `boolean esSimétricoHorizontalmente(Gato otro)`
- `boolean esSimétrico90(Gato otro)`
- `boolean esSimétrico180(Gato otro)`
- `boolean esSimétrico270(Gato otro)`

Cada método se encuentra especificado dentro de los archivos `Gatos.java` y `Gato.java`.

Punto extra

Si lo desean pueden extenderse e implementar una función de dispersión (*hash*) para cada estado generado, e implementar una lista cerrada. De esta manera se evita expandir rutas que ya se habían expandido anteriormente. Podrán obtener hasta **2** puntos extras pero cuidado: ¡no es nada trivial completar este ejercicio! pues gatos que son equivalentes bajo simetrías deben ser mapeados al mismo código de dispersión. No se sorprendan si no pueden resolver el ejercicio a la perfección.

Requisitos y resultados

Para evaluar y calificar la práctica es necesario que se implementen todos los métodos mencionados e indicados en el código, respetando implementar sólo lo que se pide (para evitar comportamientos extraños de la simulación). Es completamente válido utilizar bibliotecas adicionales si lo consideran necesario, así como la creación y uso de sus propios métodos auxiliares si lo desean. Si crean métodos auxiliares, no olviden documentar cuál es su función.

Deben correr su simulación de los espacios de estados del Gato sin simetrías, y después con ellas. Agreguen en su archivo `readme` un pequeño párrafo detallando sus observaciones con respecto a lo anterior.

3 | Algoritmos Genéticos

Rodrigo Eduardo Colín Rivera

Objetivo

Conocer el funcionamiento de los algoritmos genéticos, su aplicación, pros y contras.

Entender los mecanismos de herencia, mutación, selección y cruce, así como su relación con la selección natural. Implementar un algoritmo genético que resuelva y optimice un problema dado.

Introducción

Un algoritmo genético es una heurística de búsqueda que está basada en el proceso de **selección natural**. Su objetivo es encontrar soluciones óptimas a problemas combinatorios. La selección natural es un proceso gradual mediante el cual las características biológicas se vuelven más o menos comunes en una población. Esto depende de las características heredadas y de la diferencia de éxito reproductivo de los organismos que interactúan con su entorno. La selección natural es el mecanismo clave de la **evolución**. El término de “selección natural” fue popularizado por Charles Darwin desde el año de 1859.

Metodología

En un algoritmo genético, una **población** de candidatos a solución (también llamados **individuos** o fenotipos) es evolucionada para obtener soluciones óptimas del problema. Cada individuo tiene una representación (sus cromosomas o genotipo) que puede ser alterada o mutada; tradicionalmente, las representaciones son cadenas binarias de 0's y 1's, pero otras representaciones son posibles.

La evolución es un proceso iterativo mediante el cual se generan individuos aleato-

riamente para crear otra población en cada iteración, llamada **generación**. En cada generación, la **aptitud** (*fitness*) de cada individuo es evaluada. Usualmente la aptitud es el valor de la función objetivo del problema de optimización que se quiere resolver. Se seleccionan de manera aleatoria individuos de la población para que modifiquen su genoma, **recombinando** y posiblemente **mutando** aleatoriamente sus componentes para formar una nueva generación. La nueva generación de posibles soluciones es usada en la siguiente iteración del algoritmo. Comúnmente, el algoritmo termina cuando se alcanza el número máximo de iteraciones o el valor de aptitud de un individuo se ha aproximado lo suficiente a un valor óptimo.

Requisitos del algoritmo genético

1. Una **representación** genética del dominio de la solución.
2. Una **función de aptitud** (*fitness*) a evaluar sobre el dominio de la solución.

La representación estándar de cada individuo es un arreglo de bits, sin embargo, arreglos de otro tipo y estructuras funcionan esencialmente de la misma manera. El motivo por el cual se prefiere la representación estándar es porque facilita las operaciones de recombinación debido a la longitud fija del arreglo, también la operación de mutación se vuelve trivial como se notará más adelante.

Etapas del algoritmo genético

Inicialización

Usualmente se genera una cantidad de soluciones posibles de manera aleatoria para formar la población inicial. El tamaño de la población depende del problema y puede llegar a contener cientos de miles de individuos.

Selección

La selección es una etapa del algoritmo genético en la cual se selecciona un individuo de la población que después será recombinado con otro.

Existen diferentes maneras de realizar el proceso de selección, el más común es la **selección proporcional de aptitud** (selección de ruleta). En este tipo de selección, la aptitud (o *fitness*) del individuo se asocia con su probabilidad de selección.

Entonces si f_i es la aptitud del individuo i en la población, la probabilidad de ser seleccionado es:

$$p_i = \frac{f_i}{\sum_{j=1}^N f_j}$$

donde N es el número de individuos en la población.

Esto es similar a imaginar una ruleta de casino. Usualmente una proporción de la rueda de la ruleta es asignada a cada posible individuo basado en su aptitud. Esto se logra al dividir la aptitud de cada individuo entre el total de aptitud de todos los individuos, que es equivalente a normalizarlos a 1. Entonces un individuo es aleatoriamente seleccionado de la manera en que lo haría una ruleta que está girando.

Operadores genéticos

Recombinación La recombinación es el operador genético que permite la variación de cromosomas de los individuos de la población de una generación a otra. Es análoga a la reproducción y la recombinación biológica. El proceso de recombinación consiste en tomar más de un individuo y producir un nuevo hijo o solución.

Hay varias técnicas de recombinación, la más habitual es la recombinación de un punto y consiste en seleccionar aleatoriamente un mismo punto de corte dentro de la cadena de cromosomas de los padres e intercambiar su contenido para generar nuevos hijos:

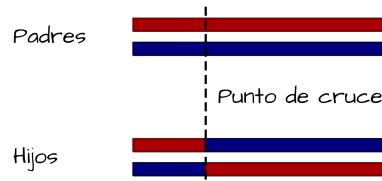


Figura 3.1 Operación genética recombinación de un punto.

Padre 1	1 1 0 1 0 0 1 0 0 1 1 0 1 1 0
Padre 2	1 0 0 1 1 1 0 1 1 0 0 1 0 1 1
Hijo 1	1 1 0 1 1 1 0 1 1 0 0 1 0 1 1
Hijo 2	1 0 0 1 0 0 1 0 0 1 1 0 1 1 0

Tabla 3.1 Ejemplo numérico de recombinación.

Mutación La mutación es el operador genético que mantiene la diversidad genética de una generación a otra, de manera análoga a la mutación biológica. La mutación altera uno o más de los genes (valores) en el cromosoma. Esta alteración puede cambiar por completo la solución antes de aplicar el operador de mutación y puede llegar a obtener mejores soluciones (o peores) dentro del algoritmo genético.

Las mutaciones ocurren valor por valor en un individuo de acuerdo a una probabilidad de mutación que es definida por el usuario. Esta probabilidad debería ser baja porque si se establece un probabilidad de mutación muy alta el algoritmo genético se convierte en una búsqueda de soluciones de manera aleatoria.

Cromosoma original	1 1 0 1 0 0 1 0 0 1 1 0 1 1 0
Cromosoma mutado	1 1 0 1 1 0 1 0 0 1 1 0 1 0 0

Tabla 3.2 Ejemplo del operador de mutación.

Terminación del algoritmo

Las iteraciones del algoritmo genético se terminan hasta que se alcanza alguna de las condiciones de terminación, que pueden ser:

- Una solución es encontrada tal que satisface un criterio mínimo.
- Un número fijo de generaciones ha sido alcanzado.
- Se alcanza el máximo de los recursos posibles (por ejemplo: tiempo de procesamiento).
- Se alcanza el valor de aptitud más alto posible o se llega a un estado en el cual las iteraciones sucesivas no producen mejores resultados (puede deberse a la falta de diversidad, por ejemplo).
- Al hacer una inspección manual.
- O combinaciones de las anteriores.

Elitismo

Hay muchas variaciones que pueden hacerse a un algoritmo genético, una de ellas es el proceso de elitismo. Esta variante permite que algunas de las mejores soluciones pasen a la siguiente generación sin alteraciones por recombinación ni mutación.

Desarrollo e implementación

Se desea resolver el problema de las ocho reinas mediante algoritmos genéticos. Este problema consiste en colocar ocho reinas del juego de ajedrez en un tablero de 8×8 de tal manera que no se ataquen mutuamente. Entonces, encontrar una solución requiere que entre cualesquiera dos reinas no comparten columna, fila, ni diagonal entre ellas. El

3. Algoritmos Genéticos

problema de las ocho reinas es un caso particular de otro más general, el de las n -reinas, que consiste en colocar n reinas en un tablero de dimensiones $n \times n$.

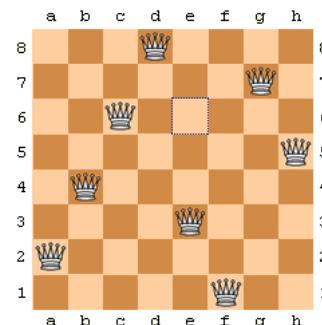


Figura 3.2 Ejemplo de solución al problema de las ocho reinas.¹

Consideraciones de la implementación

Representación genética

Es recomendable emplear una representación de un tablero mediante una arreglo de números que identifica la posición por filas de cada reina. Esta representación es muy conveniente porque evita que las reinas se ataquen por filas y facilita el proceso de encontrar una solución.

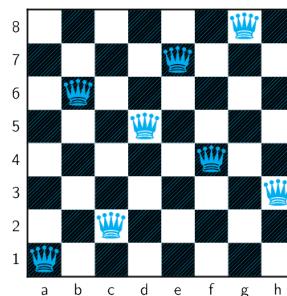


Figura 3.3 La representación del tablero sería [1, 6, 2, 5, 7, 4, 8, 3].

Función de aptitud

La optimización que se desea es encontrar un tablero en donde las reinas no se ataquen, de manera que la función de aptitud es inversamente proporcional a la cantidad de ataques

¹<https://matteoredaelli.wordpress.com/2009/01/05/n-queens-solution-with-erlang/>

entre reinas en un tablero. La figura 3.3 muestra un tablero donde sólo existe un ataque entre reinas, casi es un tablero óptimo por lo que el resultado de la función de aptitud debe ser un valor alto.

Operador de recombinación

Se utilizará el operador de corte de un punto eligiendo aleatoriamente un punto de corte, ilustrado con el siguiente ejemplo:

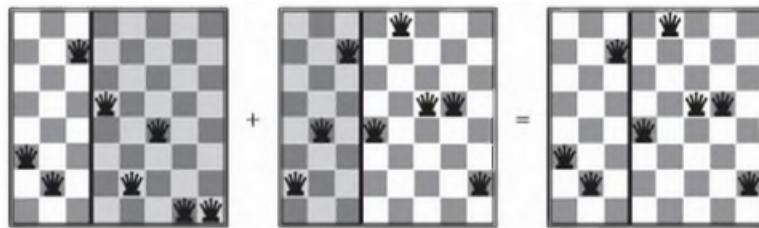


Figura 3.4 Recombinación de dos tableros para producir un nuevo tablero hijo.

Operador de mutación

El operador de mutación será definido con una probabilidad de mutación de 0.2. Es decir, se recorrerá cada gen del cromosoma (cada número del arreglo) y se modificará su valor con probabilidad 0.2.

Tablero original	[8, 3, 7, 4, 2, 5, 1, 6]
Tablero mutado	[8, 3, 7, 4, 2, 3, 1, 6]

Tabla 3.3 Ejemplo de mutación para un tablero de ajedrez.

Selección

Se utilizará el método proporcional de selección por ruleta descrito anteriormente.

Terminación del algoritmo

El algoritmo genético debe terminar cuando encuentra una solución óptima (sin ataques entre reinas) o cuando se hayan alcanzado 1000 generaciones.

Cantidad de población

La población de cada generación estará constituida por 50 individuos.

3. Algoritmos Genéticos

Elitismo

Para asegurarnos de mantener al menos una solución lo suficientemente buena, se utilizará elitismo de 1 individuo en cada generación.

Pseudocódigo

El comportamiento general del algoritmo genético puede representarse a través del siguiente pseudocódigo:

Algoritmo 1 Algoritmo genético

```
población ← newPoblación(50)
población.asignarAptitud()
while not límiteDeGeneraciones or óptimoEncontrado do
    nuevaPoblación.add(población.elitismo(1))
    while not nuevaPoblación.llena do
        individuo1 ← población.selecciónRuleta()
        individuo2 ← población.selecciónRuleta()
        hijo ← recombinación(individuo1, individuo2)
        hijo.mutación()
        nuevaPoblación.add(hijo)
    end while
    población ← nuevaPoblación
    población.asignarAptitud()
end while
print(población.mejorIndividuo())
```

Requisitos y resultados

La implementación debe mostrar cada 50 generaciones el mejor individuo encontrado y mostrar la solución óptima una vez terminado el algoritmo.

```

luis@ubuntulap: ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src
File Edit View Search Terminal Help
luis at ubuntulap in ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src
java AG
Mejor solucion en iteración 50 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 100 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 150 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 200 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 250 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 300 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 350 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 400 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 450 es : [2, 8, 3, 7, 3, 1, 6, 4, ] | fitness : 79
Mejor solucion en iteración 500 es : [7, 5, 7, 1, 6, 8, 2, 4, ] | fitness : 79

luis at ubuntulap in ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src

```

Figura 3.5 Ejemplo de resultado del algoritmo genético, donde se llegó al límite de generaciones.

```

luis@ubuntulap: ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src
File Edit View Search Terminal Help
luis at ubuntulap in ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src
java AG
Se encontró el óptimo en la generación : 1
[4, 7, 1, 8, 5, 2, 6, 3, ] | fitness : 80
Programa finalizado

luis at ubuntulap in ~/Ayudantías/2015-2/Inteligencia Artificial/Prácticas/Práctica8/src

```

Figura 3.6 Ejemplo de resultado del algoritmo genético, donde se encontró una solución antes de llegar al límite de generaciones.

Lo podrán programar en Java o Python. No olviden comentar su código.

Punto Extra: Si extienden su implementación para que pueda resolver el problema de n reinas (tablero de $n \times n$) obtendrán un punto extra.

4 | Recursión y Retractación (*Backtrack*)

Rodrigo Eduardo Colín Rivera

Objetivo

Comprender el algoritmo de búsqueda con retractación (*backtrack*) y su relación con recursión. Llevar a cabo su implementación en la construcción de laberintos. Entender la forma de representar la lógica de la construcción del laberinto y su visualización.

Introducción

Retractación es un algoritmo para encontrar soluciones en problemas computacionales donde se encuentran varios candidatos parciales de solución que se van descartando conforme avanza el algoritmo o la búsqueda.

El problema más clásico de la aplicación de retractación es el problema de las **ocho reinas**, que consiste en colocar ocho reinas en un tablero de ajedrez convencional de manera que ninguna ataque a las demás. En cada paso del algoritmo se agrega una reina a una casilla y se verifica que no ataque a las **k** reinas del tablero. Cuando no se cumple esta condición, se vuelve a una solución válida previa y se continúa probando hasta que satisfaga que no se ataquen ninguna de las reinas del tablero.

La aplicación de la técnica de retractación sólo tiene sentido en problemas que involucran soluciones parciales. Hay que notar que es una manera más eficiente que el uso de fuerza bruta, ya que se descartan muchas posibilidades que no cumplen los requisitos para ser solución del problema.

Conceptualmente, retractación es similar a la búsqueda en profundidad en árboles. Considerando que cada nodo en el árbol de búsqueda es una posible solución parcial del problema, la forma de recorrerlo es mediante recursión; podando o descartando subárboles que no son válidos como solución.

Desarrollo e implementación

Se desarrollará una aplicación que genera laberintos usando una interfaz gráfica.

Algoritmo de construcción del laberinto

El objetivo de este algoritmo es construir un escenario que consiste en una serie de pasillos que cumplan con los requisitos siguientes:

- Es posible llegar a cualquier punto del laberinto desde cualquier otro punto. Es decir, no hay regiones cerradas inaccesibles.
- La ruta entre dos puntos cualquiera del laberinto es única.

Por lo tanto, es posible seleccionar un punto como *inicio* y cualquier otro como *destino* y existe un único camino entre ellos.

Para definir este problema se empleó el ejemplo de la siguiente página para ver la construcción (primer ejemplo: “*recursive backtracker*”): [Recursive Backtracker](#)¹ También hay una liga sobre el algoritmo y una implementación hecha en lenguaje Ruby: [Maze generation](#)²

A continuación se explica la construcción del laberinto, con un algoritmo que satisface los requisitos anteriores:

1. El algoritmo comienza con un tablero de celdas de tamaño $N \times M$. Figura 4.1

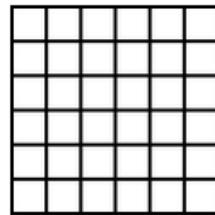


Figura 4.1 Ejemplo con $N=6$ y $M=6$.

2. Se elige una celda aleatoriamente como la celda *inicial/actual*, se marca como visitada y se agrega a la pila. Figura 4.2
3. Se elige de manera aleatoria una dirección hacia donde moverse, esto consiste en elegir una casilla adyacente que no haya sido visitada. Dependiendo de la dirección elegida se debe borrar la pared correspondiente. Se marca como visitada la celda, se empuja a una pila de celdas y se actualiza la celda actual. Figura 4.3

¹<http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap>

²<http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking>

4. Retractación

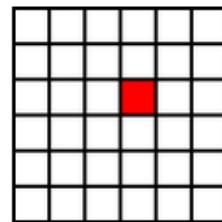


Figura 4.2 Celda inicial.

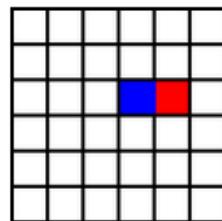


Figura 4.3 Ejemplo donde se elige la celda de la derecha. Se borra la pared entre las celdas y se marca la nueva celda actual (color rojo).

4. El paso anterior se repite hasta que ya no haya direcciones por elegir, es decir, la celda actual se queda encerrada. Figura 4.4

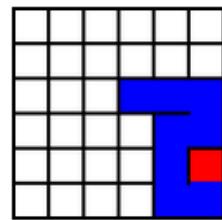


Figura 4.4 Tras algunos movimientos, la celda actual ya no puede seguir moviéndose.

5. Estando encerrados sin poder elegir una celda adyacente sin visitar, se procede a expulsar una celda de la pila para cambiar la posición de la celda actual y repetir el algoritmo desde el paso 3 hasta recorrer todo el tablero de celdas. Figura 4.5 y Figura 4.6

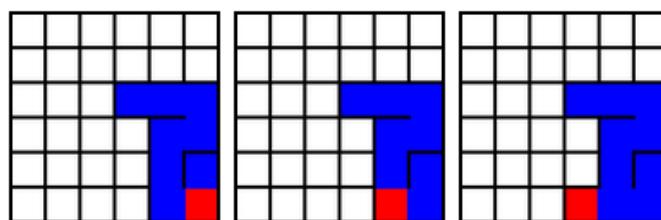


Figura 4.5 Ejemplo de un retroceso usando la pila y cambiando la celda actual.

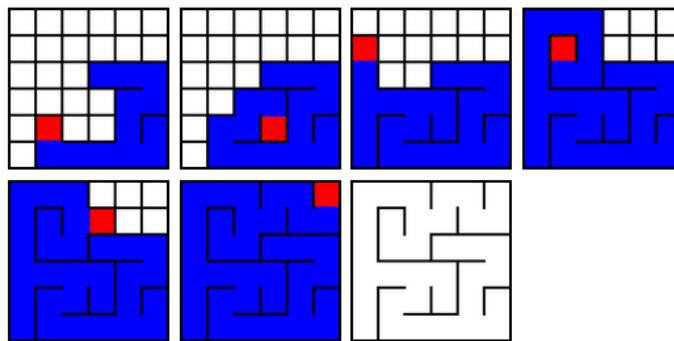


Figura 4.6 Ejemplo tras varios pasos del algoritmo hasta cubrir todo el tablero.

Implementación

Se debe implementar tanto la lógica de construcción del laberinto como su visualización. Utilicen Processing/Java ya que solo requiere usar las primitivas de dibujo: `stroke()` y `line()`. Adicionalmente también pueden usar: `fill()` y `rect()`. Es recomendable usar los siguientes métodos para una adecuada visualización: `background()` y `size()`. El material auxiliar para esta práctica ya incluye cómo dibujar el laberinto.

Requisitos y resultados

Para llevar a cabo la evaluación de esta práctica es necesario implementar la construcción del laberinto usando retractación. Empleen como estructura auxiliar durante la construcción una pila (*stack*).

La implementación debe ser lo más generalizada y robusta posible, es decir, se deben definir parámetros o valores para determinar el ancho y largo del laberinto.

No olviden documentar su código. Utilicen el estándar de JavaDoc, describan de manera breve, clara y concisa el funcionamiento de sus métodos. Si omiten documentación en su código les afectará negativamente en su calificación.

El ejemplo visto en laboratorio se construye en tiempo de ejecución. Es deseable que se muestre esta construcción pero no es necesaria. Es válido mostrar el resultado final aunque no se visualice la construcción.

Si todo se implementa correctamente debe ser posible generar diferentes tamaños de laberintos.

5 | Recocido simulado

Benjamín Torres Saavedra
Verónica Esther Arriola Ríos

Objetivo

Que el alumno se familiarice con la estrategia de mejoramiento iterativa para la resolución de problemas mediante búsquedas en el espacio de estados, conocida como recocido simulado.

Introducción

Recocido simulado

Este algoritmo puede ser visto como una mejora al algoritmo de ascenso de colinas. Ascenso de colinas comienza con la propuesta de una solución parcial o no óptima a un problema, la cual mejora de manera iterativa hasta encontrarse con la óptima o estancarse en un máximo local. Recocido simulado integra una selección de soluciones estocástica, es decir, para elegir la siguiente solución no siempre escoge a la mejor vecina, con lo cual se le da la libertad de explorar zonas distintas del espacio de soluciones, en las cuales el ascenso de colinas puede detenerse fácilmente.

El pseudocódigo de este algoritmo es descrito en el libro de Russell y Norving [2010](#) en la sección de algoritmos de mejoramiento iterativo y lo podemos ver en el Algoritmo 2.

Problema del agente viajero (TSP)

Este problema consiste en encontrar, dada una lista de ciudades, una ruta que pueda seguir un agente para recorrer todas las ciudades y volver a aquella en la cual inició el

Algoritmo 2 Recocido simulado

```

function RECOCIDOSIMULADO(problema, horario)
    # problema: un problema
    # horario: mapeo de tiempo a temperatura.
    actual ← HAZNODO(Estadoinicial(problema))
    for t ← 1 a  $\infty$  do
        T ← horario(t)
        if T <  $\varepsilon$  then
            return actual
        end if
        siguiente ← un sucesor de actual elegido al azar.
         $\Delta E \leftarrow \text{VALOR}(\text{siguiente}) - \text{VALOR}(\text{actual})$ 
        if  $\Delta E > 0$  then
            actual ← siguiente
        else
            actual ← siguiente sólo con probabilidad  $e^{\frac{\Delta E}{T}}$ 
        end if
    end for
end function

```

viaje. Para que este recorrido valga la pena debe ser lo más económico posible o, en su defecto, recorrer la menor distancia que permita visitar todas las ciudades.

Este problema es ampliamente conocido por ser de la clase NP-Completo, es decir, que no se conoce ningún algoritmo determinista que pueda resolverlo en tiempo polinomial, afortunadamente para esta práctica no usaremos un algoritmo de tal tipo y podremos aproximarnos a una solución en un tiempo razonable.

Desarrollo e implementación

Para esta práctica cuentas con código auxiliar en Java, que deberás completar con la finalidad de resolver el problema del agente viajero.

El código fuente consta de los archivos:

- `recocido/Solución.java`. Se encargará de representar una ruta del agente viajero que pase por todas las ciudades,
- `recocido/RecocidoSimulado.java`. Implementará el algoritmo anteriormente descrito.
- `recocido/DatosPAV.java`. Lee la información desde los archivos de datos con la información de las ciudades que se utilizarán.

5. Recocido simulado

- `recocido/Main.java`. Contiene el esqueleto con la idea principal de cómo utilizar el resto del código para buscar soluciones.

Implementación

Se debe programar una clase hija de `Solución`. Esta clase hija agregará los atributos del(de los) tipo(s) necesario(s) para representar una propuesta de solución al problema que se desea resolver.

En la página <http://www.math.uwaterloo.ca/tsp/world/countries.html#DJ> se pueden encontrar una serie de países con ciudades y sus correspondientes coordenadas. Se incluye un ejemplar pequeño de esta página con esta práctica para que te sirva al probar tu código. Si asumes una conectividad total de las ciudades y usas la distancia euclídea como métrica, tu tarea será tratar de aproximarte lo mejor posible a la longitud del camino óptimo para recorrer todas las ubicaciones de las ciudades en el país seleccionado. Puedes incorporar la información a tu programa como te sea más conveniente.

Los métodos a implementar dentro de una clase hija de `Solución` son:

- Un constructor.

Este método deberá inicializar una representación con una propuesta para solución de un problema, en nuestro caso, una ruta del problema del agente viajero. Deberás elegir cómo representar la solución. No es necesario que dicha solución sea correcta. Puedes modificar la firma del constructor como consideres necesario para crear un propuesta de solución inicial aleatoriamente a partir de la especificación del problema a resolver.

En el caso del agente viajero puede ser que la solución visite todas las ciudades, pero que la distancia recorrida no sea mínima y/o que las visite más de una vez. La especificación del problema está dada por los archivos `.tsp` con la información sobre las ciudades.

- `public Solución siguienteSolución()`

Genera una nueva solución perturbando de manera aleatoria al objeto que lo llama. Observa que al sobreescibir el método puedes cambiar el tipo de regreso para evitar hacer audiciones (*castings*).

- `public float evaluar()`

Califica la solución actual según la heurística elegida de acuerdo al problema a resolver.

En este caso la evaluación estará relacionada con la longitud del recorrido propuesto por la solución actual. La función de evaluación que elijas debe cumplir que el re-

corrido óptimo del problema del agente viajero satisface que $\text{óptimo.evaluar}() \leq s.\text{evaluar}()$ para toda solución posible.

Para la clase RecocidoSimulado:

- public RecocidoSimulado()

Inicializa los parámetros del algoritmo. En principio ya funciona así, pero la puedes modificar si lo consideras necesario.

- public float nuevaTemperatura()

Calcula la nueva temperatura, se espera que a lo largo de las iteraciones este valor decrezca, llegando a cero en el último paso.

- public Solución seleccionarSiguienteSolución()

Dada la solución actual, este método debe obtener una modificación suya y elegir esta nueva solución con cierta probabilidad dependiendo de si es mejor o no, según el algoritmo presentado anteriormente.

- public Solución ejecutar()

Ejecuta el algoritmo y devuelve la mejor solución encontrada.

Finalmente, en la clase Main se crea un objeto tipo RecocidoSimulado que ejecuta el algoritmo por algún número de iteraciones:

- Usa el argumento en args[0], que deberá ser la ubicación de un archivo .tsp, para cargar la descripción de una ciudad. Se te da la clase DatosPAV para facilitar esta tarea.

- Calcula los parámetros para el constructor de RecocidoSimulado:

- Crea un objeto de la clase hija de Solución que implementaste, de modo que represente una primer ruta a través de la ciudad descrita en el archivo .tsp.

- Calcula la temperatura inicial y decaimiento adecuados para ejecutar recocido simulado dependiendo del número de iteraciones.

- Modifica el ciclo para monitorear la evolución del algoritmo entre iteraciones.

Punto extra

Realiza las modificaciones que consideres pertinentes para poder utilizar una estrategia similar al recocido simulado para minimizar la siguiente función:

$$f(x, y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(\cos(2\pi x)+\cos(2\pi y))} + e + 20$$

para $-5 \leq x, y \leq 5$.

Requisitos y resultados

El código debe ser implementado de manera eficiente y estar documentado para esclarecer su funcionamiento. Además, debe encontrar una solución válida al problema del agente viajero.

6 | A* Pakuman

Verónica Esther Arriola Ríos

Objetivo

Que el alumno implemente y refuerce su comprensión del algoritmo A*.

Introducción

El algoritmo A* es el algoritmo de búsqueda en el espacio de estados que tiene garantizado encontrar la solución de menor costo explorando la menor cantidad de estados en el espacio de búsqueda. Para ello se beneficia de las propiedades del algoritmo de Dijkstra para encontrar la ruta mínima entre dos puntos de un grafo, aumentada con el uso de información del dominio a través de una función heurística, que estima la distancia faltante hasta la meta, permitiéndole así, explorar primero los caminos más prometedores.

Para que al ejecutar el algoritmo se cumplan estas características, la función heurística debe ser:

Admisible También le podemos decir **optimista**, nunca debe sobreestimar el costo hasta la meta.

Consistente Ir directamente de un estado A a un estado B debe ser igual o menos costoso, que si hay una desviación pasando por C. Esto se expresa en la desigualdad:

$$h(n) \leq c(n, a, n') + h(n')$$

Donde el punto A es el estado n y B es la meta. Se debe cumplir entonces que la distancia estimada desde n sea menor o igual que si se pasa por n' al ejecutar la acción a .

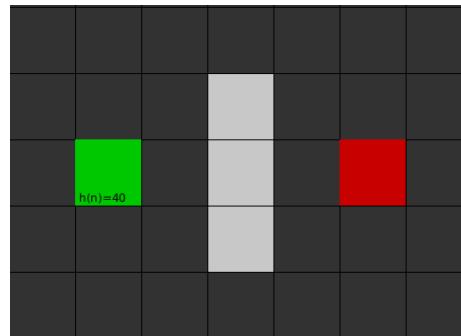


Figura 6.1 Escenario construido con mosaicos discretos. El problema de búsqueda de caminos consiste en encontrar una ruta desde el cuadro verde al cuadro rojo.

Navegación

Un curso de Inteligencia Artificial no estaría completo sin haber implementado A*. Una de las aplicaciones más populares de este algoritmo es el problema de navegación en videojuegos y robótica. Comenzaremos siguiendo el ejemplo de Patrick Lester 2003 en un mundo hecho de mosaicos.

En el ejemplo de Patrick, asumimos que tenemos un personaje que quiere ir desde un punto A hasta un punto B y que ambos puntos están separados por una pared. Este ejemplo se puede apreciar en la Figura 6.1, donde el cuadrado verde es el punto A, el rojo es el punto B y el rectángulo gris, la pared mencionada anteriormente. Curiosamente, este sencillo escenario no puede ser resuelto por ascenso de colinas, pero para A* es pan comido.

Si el mundo donde se encuentran nuestros personajes es continuo, como los ambientes donde se mueve un robot, el primer paso que debemos hacer es simplificar el área de búsqueda, dividiendo nuestro mundo en una rejilla de mosaicos. En los videojuegos algunos mundos ya son así desde un principio. En el caso de robótica se puede proceder a crear rejillas de ocupación discretizando las coordenadas del plano y representando al mundo con mosaicos ocupados/libres, semejantes a los de los videojuegos. Con esto, podremos representar el mundo con una matriz bidimensional. Cada posición en la matriz representará un mosaico del mundo, el cual será transitble o no transitble. Entonces, el objetivo del algoritmo A* es calcular a qué mosaico nos debemos mover en cada turno para lograr llegar al punto B. Una vez calculado esto, el personaje del juego se moverá del centro del mosaico donde se encuentra actualmente al centro del mosaico obtenido con A*. Observa entonces que cada mosaico del mundo es un estado posible para nuestro personaje.

Ahora pensaremos en el mundo como un grafo, el puntos central de cada mosaico corresponde a un *nodo*; de este modo los mosaicos podrían ser círculos, hexágonos o triángulos, no sólo cuadrados; cada mundo puede tener características diferentes y ser simplificado de distintas formas. Cuando dos mosaicos son adyacentes y es posible des-

plazarse de uno hacia el otro, decimos que hay un arista conectándolos. Así, el algoritmo que busca la ruta más corta entre el punto A y el punto B es un recorrido sobre un grafo.

Iniciando la búsqueda

Después de haber simplificado el área de búsqueda a un grafo con nodos y aristas, el siguiente paso es dirigir una búsqueda para encontrar el camino más corto. Comenzamos por el punto A, luego consideramos los cuadros adyacentes (estados sucesores) para ver en qué dirección movernos y continuamos alejándonos hasta que encontremos nuestro destino. Para ello utilizaremos dos estructuras a las cuales llamaremos *lista abierta* y *lista cerrada*.

Empezamos la búsqueda haciendo lo siguiente:

1. Añadimos el punto inicial A a la **lista abierta**. Esta lista contiene los cuadrados que vamos a considerar para que formen parte del camino. Si, por azares del destino, el punto A es igual al punto B, terminamos nuestra ejecución con un plan vacío: el personaje no necesita hacer nada, ya está en su destino.
2. Sacamos el cuadro inicial A de la lista abierta y lo añadimos a una **lista cerrada** de cuadrados que no necesitan ser considerados de nuevo.
3. Nos fijamos en todos los cuadrados alcanzables o transitables adyacentes al punto de inicio, ignorando cuadrados no transitables (por contener muros, muebles, agua, precipicios, etc.). Se añaden a la lista abierta, marcando que el punto A es su **cuadrado padre**, pues estamos considerando llegar desde ahí. El cuadrado padre será muy importante para trazar nuestro camino al final de la búsqueda. La estructura dentro de la cual guardaremos una referencia al cuadrado vecino y a su padre, y que ofreceremos a la lista abierta, se llama **nodo de búsqueda**.

En este punto, se tendrá algo como la figura 6.2, en un momento explicaremos el significado de los números en las casillas. En este diagrama, el cuadrado verde es el **estado inicial** y ya ha sido añadido a la lista cerrada. Todos los cuadrados adyacentes están ahora en la lista abierta para ser considerados. Cada uno tiene un indicador que señala a su padre, que es el cuadro inicial.

Ahora necesitamos elegir a uno de los cuadrados de la lista abierta para explorar esa ruta. Elegimos aquel vecino que tenga costo estimado más bajo, es decir, aquel al que hemos llegado por la mejor ruta y promete estar más cerca de la meta, hasta donde sabemos, para ello calcularemos la función $f(n)$.



Figura 6.2 Mosaicos en la lista abierta con referencia a su padre.

Puntuando el camino

Para determinar qué cuadrado exploraremos a continuación usaremos la siguiente ecuación:

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

donde:

- $g(n)$ es el costo del movimiento para ir desde el punto inicial A al cuadro n de la rejilla, siguiendo el mejor camino que se generó para llegar a n .
- $h(n)$ es el costo estimado del movimiento para ir desde ese cuadro n hasta el destino final, el punto B. Si conociéramos ese costo exactamente no necesitaríamos buscar la mejor ruta, bastaría con movernos hacia el vecino con el valor más bajo, en lugar de ello sólo contaremos con una aproximación, a esta se le llama **heurística**.

Para generar el camino completo de A a B, tendremos que ir sacando a los nodos de la lista abierta con menor $f(n)$ hasta llegar a la meta. Sin embargo, calcular esta función tiene su chiste, pues su valor puede cambiar conforme exploramos. Veamos entonces cómo se hace ese cálculo.

Heurística

La primer función que podemos calcular es $h(n)$, pues su valor no cambia una vez dado el ambiente. Lo óptimo es calcularla la primera vez que el algoritmo ve al cuadrado n , cuando crea el nodo búsqueda y lo inserta a la lista abierta.

Cuando se trata de problemas de desplazamiento, como es el caso aquí, una aproximación utilizada frecuentemente es la distancia Euclíadiana (la ruta más corta, si no hubiera obstáculos, sería la línea recta), aunque tiene la desventaja de involucrar el cálculo de una raíz cuadrada. Por ello existen otras aproximaciones, que pueden ser más o menos adecuadas, dependiendo del ambiente.

Además de la distancia Euclíadiana, otra técnica para aproximar distancias es el método Manhattan, donde se calcula el número total de cuadros movidos horizontalmente y verticalmente para alcanzar el cuadrado destino desde el cuadro actual, sin hacer uso de movimientos diagonales e ignorando cualquier obstáculo. Se llama método Manhattan porque está inspirado en la ciudad que lleva ese nombre, donde las manzanas están diseñadas en forma de rejilla y para ir desde un lugar a otro no es posible tomar atajos atravesando una manzana en diagonal. Observa que esta estimación es mejor si el agente no puede moverse en diagonal, pues de este modo nunca sobreestimará qué tan lejos está la meta.

En esta práctica asignaremos un costo de 10 por cada movimiento vertical u horizontal a través de un cuadrado y un costo de 14 para un movimiento diagonal (una aproximación razonable de $\sqrt{2} \times 10$), pues usar números enteros es mucho más rápido para la computadora y la diferencia en tiempo puede ser significativa si el algoritmo se ejecutará muchas veces sobre muchos nodos.

Las puntuaciones $h(n)$ se calcularán estimando la distancia Manhattan desde n hasta el cuadrado objetivo (en rojo), moviéndose solo horizontal y verticalmente e **ignorando el muro** que está en el camino. Usando este método, en la Figura 6.2 el cuadro a la derecha del inicial, está a 3 cuadros del cuadrado rojo lo que da una puntuación $h(n)$ de 30. Los cuadros arriba y abajo del inicial están a sólo 4 cuadros de distancia (tomando sólo movimientos en dirección horizontal y vertical), dando una puntuación $h(n)$ de 40.

Costo parcial

La siguiente función a estimar es $g(n)$, el costo del movimiento para ir desde el punto de inicio al cuadro n , usando el camino generado para llegar ahí. Observa que por lo mismo, solamente se puede calcular conforme se van eligiendo los nodos durante la búsqueda.

Una vez calculado el costo $g(n)$, el costo $g(n')$ para su vecino n' , al que estamos llegando, es $g(n)$ más 10 o 14 dependiendo de si el movimiento para llegar a él es ortogonal o se realiza en diagonal desde su cuadro padre.

En la Figura 6.2 se pueden ver estos cálculos. En el cuadrado a la derecha del inicial $g(n) = 10$. Esto es debido a que está solo a un cuadro del cuadrado inicial en dirección horizontal. Los cuadros inmediatamente encima, abajo y a la izquierda del cuadrado inicial tienen todos el mismo valor $g(n)$ de 10. Los cuadros diagonales tienen un valor $g(n)$ de 14. $f(n)$ se calcula sumando $g(n)$ y $h(n)$.

Continuando la búsqueda

Para continuar la búsqueda, se elige al nodo con la puntuación $f(n)$ más baja de todos aquellos que estén en la lista abierta. Para que esta operación se realice lo más eficientemente posible, la lista abierta se debe implementar con una cola de prioridades. Después hacemos lo siguiente con el cuadro seleccionado:

1. Lo sacamos de la lista abierta y lo añadimos a la lista cerrada.
2. Comprobamos todos los cuadrados adyacentes:
 - a) Ignorando aquellos que estén en la lista cerrada o que sean intransitables o a los que no se puede pasar: terrenos con muros, agua o cualquier terreno prohibido.
 - b) Añadimos los cuadros a la lista abierta si no están ya en esa lista. Hacemos que el cuadro seleccionado sea el **padre** de los cuadros nuevos.
 - c) Si el cuadro adyacente ya está en la lista abierta, comprobamos si el camino nuevo a ese cuadro es mejor que el que tenía, es decir, si el valor de $g(n)$ con este padre es menor que el que se había estimado con su padre anterior. Si no es así, no haremos nada. Por otro lado, si el costo $g(n)$ del nuevo camino es más bajo, cambiamos el padre del cuadro adyacente al cuadro seleccionado (en el diagrama superior, cambia la dirección del puntero para que señale al cuadro seleccionado). Finalmente, recalculamos $f(n)$ y $g(n)$ para ese cuadrado.

A continuación se muestran dos versiones de la ejecución completa del algoritmo para un escenario ligeramente más complejo. En la primer versión usamos distancia Manhattan. La Figura 6.3 muestra el contenido final de las estructuras auxiliares, las listas cerrada y abierta y los valores calculados de h , g y f . Observa que, para este mapa donde se permiten movimientos en diagonal, esta heurística no es **admissible**, pues la ruta más corta puede utilizar diagonales y, entonces, estar más cerca de lo que estima. La consecuencia es que A* no siempre encuentra la ruta óptima en estas condiciones. La Figura 6.4 muestra lo que sucede cuando se usa la distancia Euclíadiana, que sí es una heurística admisible.

Desarrollo e implementación

Para esta práctica, considérate un miembro nuevo en un compañía de programación de videojuegos. El resto del equipo ha estado trabajando en un *remake* de Pacman con JavaFX y a ti te han asignado la siguiente tarea: programar al más férreo perseguidor de Pacman, el fantasma rojo **Sombra**, mejor conocido como Blinky. Para ello, has decidido utilizar el mejor algoritmo para mundos finitos deterministas: A*.

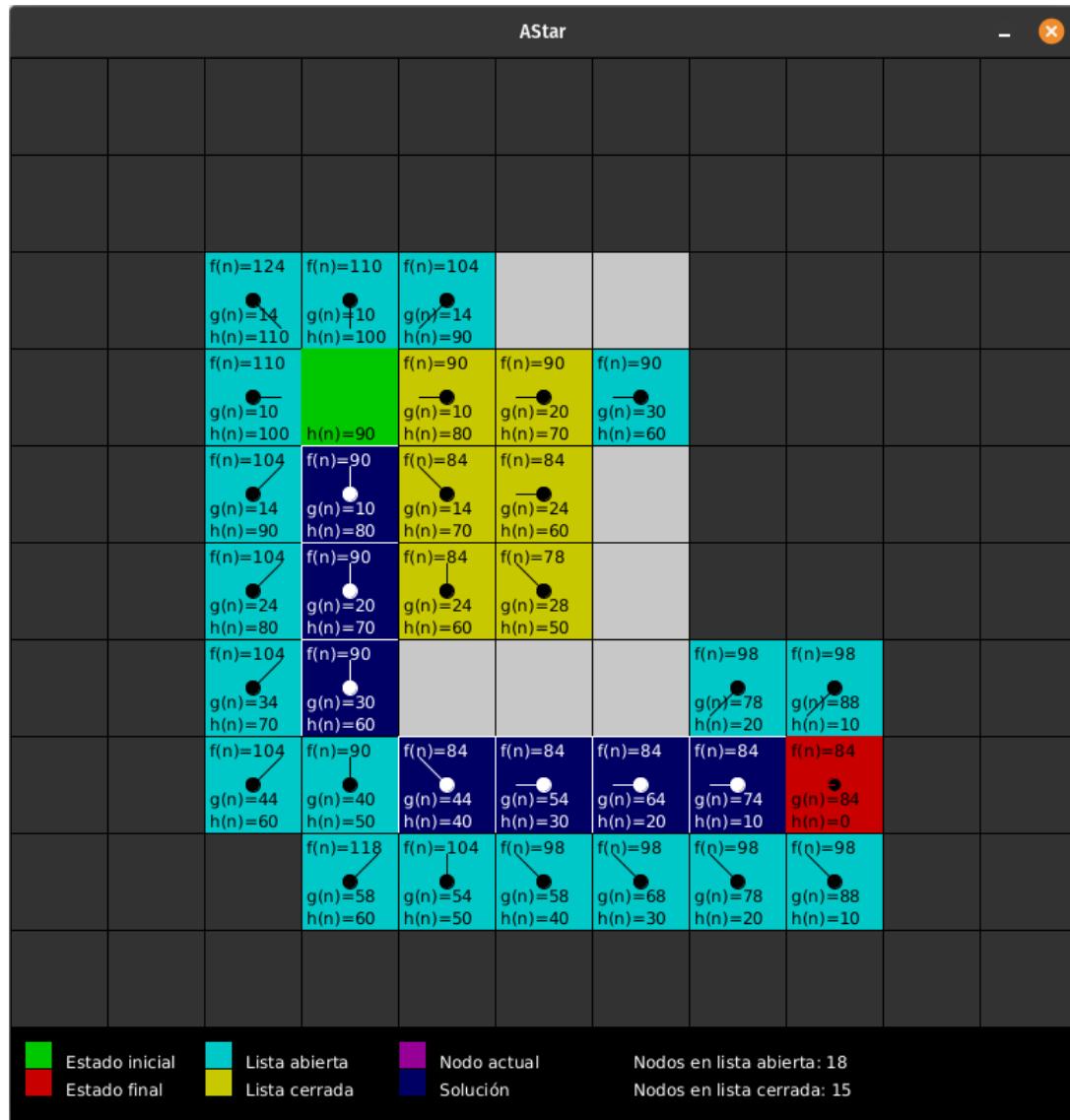


Figura 6.3 Ejecución completa del algoritmo con distancia Manhattan. A* encuentra una ruta, pero no es la óptima.

Para realizar esta tarea se te entrega el código con el escenario listo para jugar y la documentación que han elaborado tus colegas. El código aún no está terminado, pero ya incluye como muestra el primer nivel del juego y está listo para añadirle archivos de configuración con cualquier otro nivel. Igualmente el código para controlar a Pacman usando las flechas del teclado ya funciona. El único detalle es que Sombra aún no persigue a Pacman, se mueve siempre en una dirección fija que, lo notarás de inmediato, no lo lleva muy lejos.

El programa está compuesto por varios paquetes y clases pero como está diseñado con orientación a objetos, sólo necesitas entender algunas clases para agregar el algoritmo que necesita Sombra. El diagrama UML de la Figura 6.5 te muestra los paquetes, clases,

6. A* Pakuman

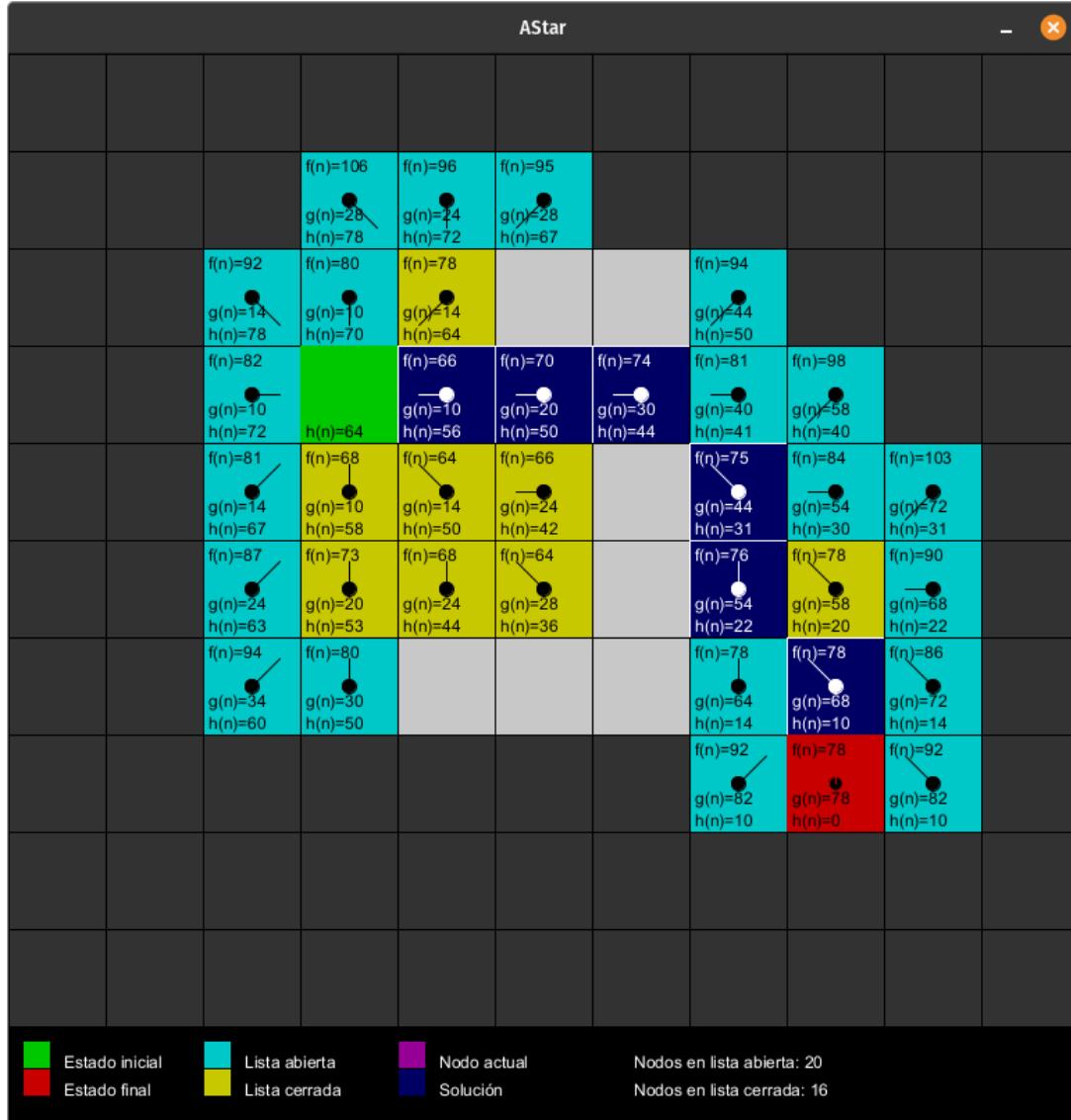


Figura 6.4 Ejecución completa del algoritmo con distancia Euclidiana. A* encuentra la ruta óptima.

atributos y métodos que son relevantes para tu tarea.

Implementación

Tu trabajo consiste en implementar la función:

```
LinkedList<Movimiento> resuelveAlgoritmo(Estado estadoInicial, Estado estadoFinal)
```

de la clase pacman.personajes.navegacion.AEstrella. La variable estadoInicial

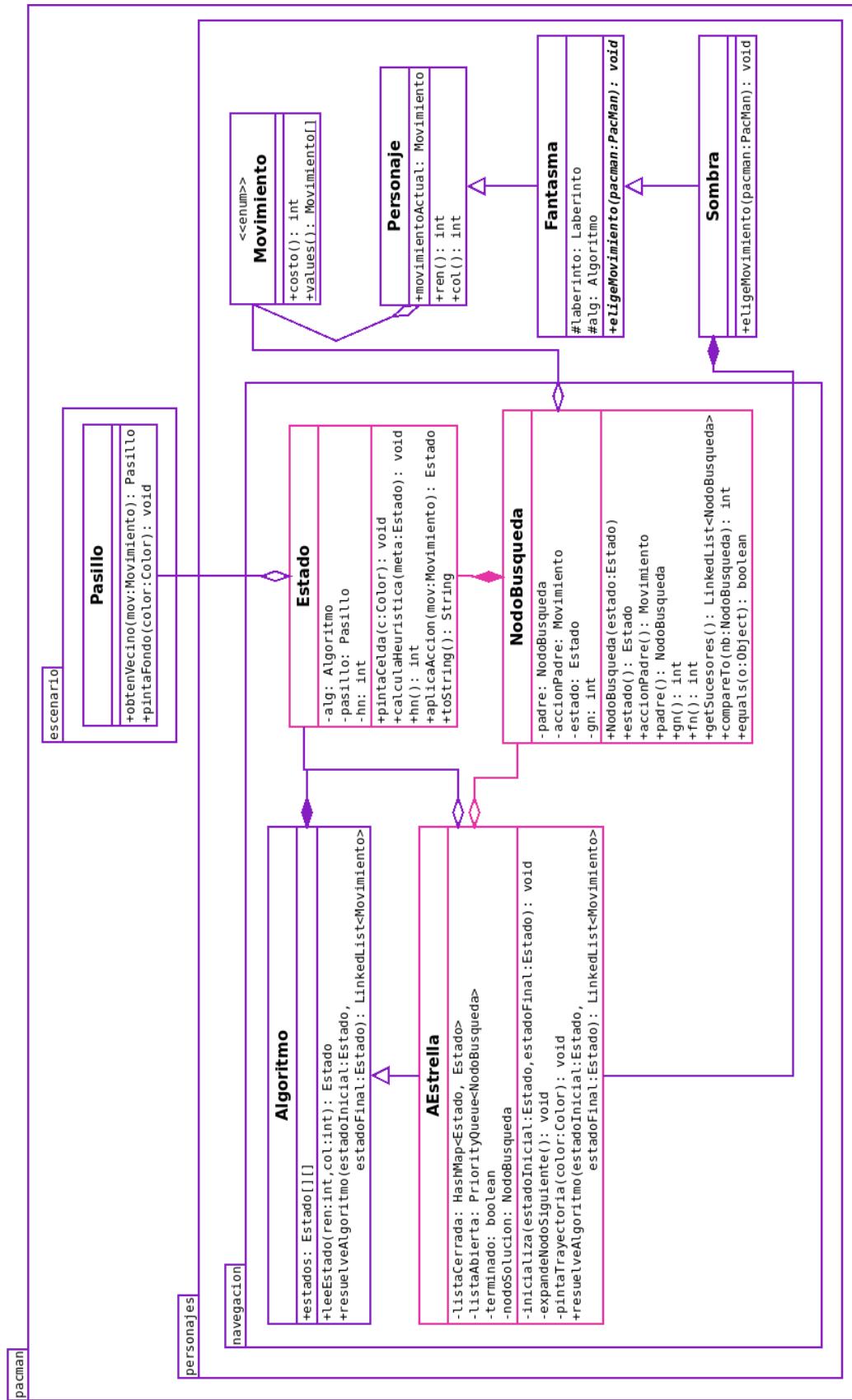


Figura 6.5 UML con las clases relevantes para programar el algoritmo de navegación para Sombra. Las clases con métodos a implementar se muestran en rosa.

6. A* Pakuman

contiene información con las coordenadas de la casilla inicial, mientras que `estadoFinal` indica a qué casilla se quiere ir. El trabajo de esta función es devolver la secuencia de acciones que se deben realizar (movimientos) hasta llegar a la casilla final.

Para que ubiques mejor cómo será usada esta función te informan que, de momento, el `estadolInicial` corresponde a la posición actual de Sombra y `estadoFinal` es la posición de Pacman. Sombra manda llamar esta función en cada tick del reloj con las nuevas posiciones suya y de Pacman. De hecho, lo único que se utiliza es el primer movimiento de la trayectoria... pero no te confíes, las cosas pueden cambiar, otras optimizaciones podrían ser introducidas y por lo tanto debes calcular la lista completa de movimientos. Aunque estos detalles son irrelevantes para tu código, saber cómo será usado te ayudará a visualizar lo que necesitas hacer. Sin embargo, observa que por ser general, esta función podría llegar a utilizarse no sólo para mover a Sombra sino que, en algún momento, otro fantasma con otro comportamiento también podría aprovecharla. Por ejemplo, el encargado de las emboscadas **Pinky** podría usar A* para llegar a la esquina más cercana a Pacman enviando como parámetros sus coordenadas y las de dicha esquina. Por lo tanto, no debes usar información de más o de menos confiándote en el estado actual de la aplicación, ni afectar el resto del juego con intervenciones innecesarias para que tu código sea reutilizable con facilidad.

Para ayudarte a realizar esta tarea el paquete te ofrece las herramientas siguientes:

1. La clase `Estado` pertenece al subpaquete `navegación`. Es una especie de envoltura con una referencia a un `Pasillo` por donde puede pasar el fantasma. Un objeto de este tipo puede almacenar la información de ejecución relevante para el algoritmo A* como es el valor de la función $h(n)$. De este modo te brinda acceso al estado en el mundo (el laberinto donde se encuentran nuestros personajes) y te brinda un espacio para no afectar al mundo con información que sólo concierne al algoritmo de navegación de un fantasma.
2. La superclase `Algoritmo` se encarga de inicializar, tras la creación del laberinto, un arreglo de objetos tipo `Estado`. De este modo cuentas ya con un objeto `estado` por cada pasillo en el laberinto, en el mismo renglón y columna que el pasillo en el mundo. Ojo, los valores de la heurística $h(n)$ cambian cada vez que cambia el lugar al que quiere moverse el fantasma, así que recuerda recalcular tantos valores como sean necesarios cada vez que ejecutes el algoritmo con metas distintas.
3. La clase `NodoBusqueda` contiene la información del grafo de búsqueda que va generando A* conforme explora los estados. Por eso su primer atributo es una referencia al estado que está explorando. También aquí se almacena el valor de $g(n)$, pues este valor depende de la ruta que se siga para llegar al estado n desde el estado inicial; por ello mismo incluye la referencia al nodo padre y la acción que se realizó para llegar al estado que contiene. Recuerda que, en general, varios nodos de búsqueda podrían apuntar al mismo estado. Como en esta práctica se realizará una búsqueda en grafo no será el caso.

4. La clase Pasillo ya representa, de hecho, la gráfica de estados por donde puede transitar un personaje, pues cada pasillo contiene referencias a sus pasillos vecinos. Puedes acceder a ellos mediante el método `obtenVecino(Movimiento mov)`. Te servirá para generar los estados sucesores de cada estado.

Los métodos que deberás implementar son los siguientes:

1. En la clase Estado la función:
 - a) `calculaHeuristica(Estado meta)`
2. En la clase NodoBusqueda:
 - a) `LinkedList<NodoBusqueda> getSucesores()`
3. En la clase AEstrella:
 - a) `LinkedList<Movimiento> resuelveAlgoritmo(Estado estadoInicial, Estado estadoFinal)`

se te sugiere la implementación de dos métodos auxiliares:

- a) `inicializa(Estado estadoInicial, Estado estadoFinal)`
- b) `void expandeNodoSiguiente()`

Como son métodos privados, tienes la libertad de utilizarlos o modificarlos a tu gusto. Se te provee con `pintaTrayectoria(Color color)` para ayudarte a depurar tu código.

Requisitos y resultados

Los requisitos para tu trabajo son los siguientes:

1. Para implementar el algoritmo puedes agregar atributos sólo si realmente necesitas recordar los valores de esas variables entre distintas llamadas a los métodos de la clase, únicamente en las clases marcadas con rosa.
2. Es estas clases también puedes agregar métodos auxiliares si lo consideras necesario. De hacerlo, no olvides documentar cual es su función.
3. No debes agregar ni atributos ni funciones a ninguna otra clase, ya tienes toda la información que necesitas; piensa que el resto del programa está siendo desarrollado por otros compañeros de la empresa y no tienes permiso de tocar sus componentes.

6. A* Pakuman

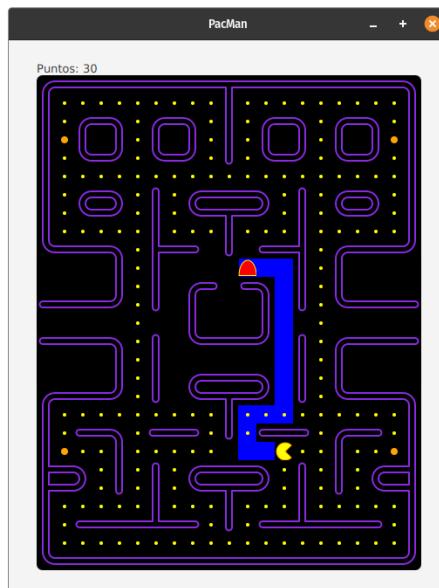


Figura 6.6 Sombra persiguiendo a Pacman por la ruta óptima.

Cuando termines tu implementación podrás ver a Sombra persiguiendo a Pacman por la ruta óptima; la única forma de alejarlo un poco será transportándote por los pasillos que sacan a Pacman por un lado de la pantalla y lo regresan por el otro. Si gustas, borrar e iluminar la ruta que sigue Sombra a cada paso es muy fácil ya teniendo la trayectoria completa.

7 | Perceptrón, unidad fundamental de las redes neuronales

Luis Alfredo Lizárraga Santos

Objetivo

Conocer el funcionamiento de la unidad elemental con la cual se construyen las redes neuronales: el perceptrón, y lograr implementar un perceptrón simple que aprenda las operaciones AND y OR para tres variables.

Introducción

“Una red neuronal se puede definir como un modelo de razonamiento basado en el cerebro humano” (Negnevitsky 2005, pág. 166). Basándonos en el libro de Negnevitsky explicaremos cómo funciona el cerebro, cómo las Redes Neuronales modelan el cerebro, cómo aprenden y finalizaremos con el Perceptrón (Negnevitsky 2005, cap. 6).

Redes Neuronales Artificiales

Sabemos que el cerebro consiste en un conjunto densamente interconectado de unidades básicas de procesamiento, llamadas neuronas. El cerebro humano contiene cerca de 86 mil millones de neuronas (Herculano-Houzel 2009) y entre 100 y 500 billones de conexiones (Drachman 2005), llamadas **sinapsis**.

A pesar de que cada neurona tiene una estructura muy simple, un conjunto (aunque sea pequeño) de estos elementos constituye un poder de procesamiento enorme. Una neurona está constituida por un cuerpo celular, llamado **soma**, un conjunto de fibras llamadas **dendritas**, y una única fibra larga llamada **axón**. La figura 7.1 representa dos neuronas conectadas.

7. Perceptrón

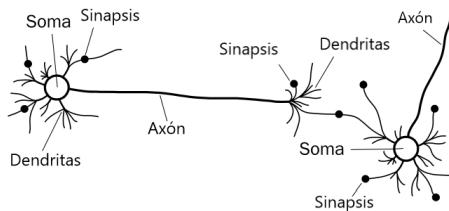


Figura 7.1 Partes de una neurona biológica. (Negnevitsky 2005, pág. 166)

Señales se propagan de una neurona a otra por medio de reacciones electroquímicas complejas. Substancias químicas que se liberan desde las sinapsis causan un cambio en el potencial eléctrico del cuerpo celular de la neurona. Cuando este potencial sobrepasa su umbral, una señal eléctrica, llamada **potencial de acción**, se manda a través del axón. Este pulso se dispersa y eventualmente llega a las sinapsis, haciendo que estas incrementen o decrementen su potencial. Pero el descubrimiento más interesante es que las neuronas exhiben **plasticidad**.

Esta plasticidad permite que las conexiones hacia las neuronas que conducen a la “respuesta correcta” se vean fortalecidas, mientras que las conexiones que llevan a la “respuesta equivocada” sean debilitadas. Como resultado, las redes neuronales tienen la habilidad de aprender mediante la experiencia.

¿Cómo modelan al cerebro?

Las neuronas de la red neuronal artificial se conectan entre sí usando conexiones con un peso dado, pasando señales de una neurona a otra. Cada neurona recibe un número de señales de entrada a través de sus conexiones, pero sólo produce una salida, como se muestra en la figura 7.2. Esta señal de salida se transmite por la conexión saliente de la neurona (lo equivalente al axón en la neurona biológica). Esta conexión saliente, a su vez, se separa en varias ramas que transmiten la misma señal. Estas ramas terminan como conexiones de entrada de otras neuronas en la red.

¿Cómo aprenden?

Las neuronas se conectan mediante enlaces que tienen un peso asociado a ellos. Estos pesos son los medios para guardar información a largo plazo. Expresan la importancia de cada entrada de la neurona. Entonces, las redes neuronales “aprenden” por medio de ajustes a estos pesos.

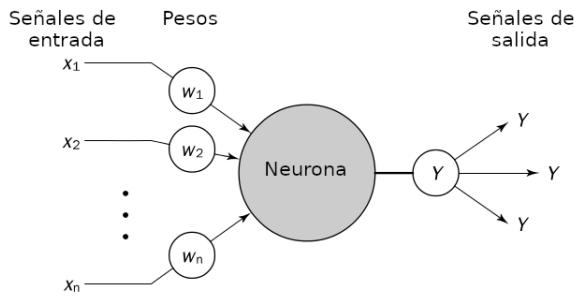


Figura 7.2 Diagrama de un perceptrón. (Negnevitsky 2005, pág. 168)

Neuronas Artificiales

¿Cómo se determina la salida?

La neurona calcula la suma de las señales de entrada multiplicadas por el peso de su conexión con la neurona siguiente (combinación lineal) y compara este resultado con un umbral θ . Si el total es menor que el umbral, la salida de la neurona es (-1), si es mayor entonces la neurona se activa y su salida es (1). A lo descrito anteriormente se le llama función de activación.

Hay varias funciones que se pueden llegar a utilizar en lugar de esta, la figura 7.3 presenta algunas de ellas.

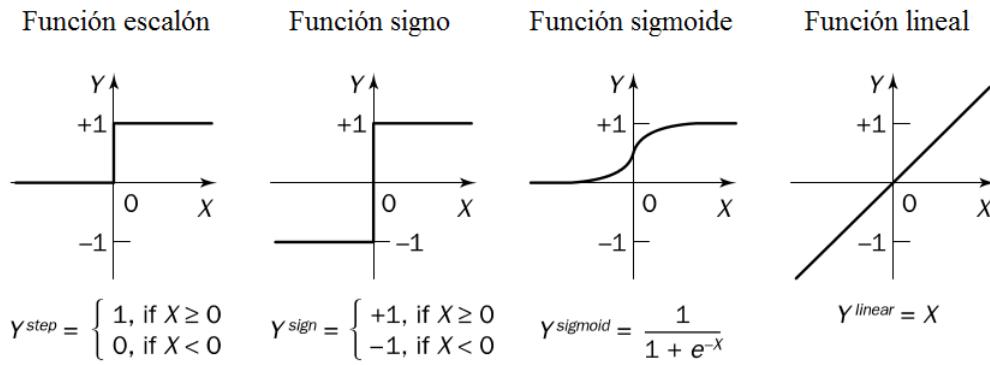


Figura 7.3 Ejemplos de funciones de activación de una neurona. (Negnevitsky 2005, pág. 169)

Perceptrón

Un perceptrón es la forma más simple de una red neuronal, ya que representa a una sola neurona.

El perceptrón consiste en un combinador lineal seguido de una función de activación.

7. Perceptrón

Se le aplica la función umbral a la suma con pesos de las entradas menos el umbral, la cual produce una salida positiva si la suma es positiva, o una salida negativa si la suma es negativa. El objetivo del perceptrón es clasificar entradas en una de dos clases. Esto se expresa con la siguiente función:

$$Y = f \left(\sum_{i=1}^n x_i w_i - \theta \right)$$

(Negnevitsky 2005, pág. 169)

Pero ¿cómo aprende a clasificar?

Esto se logra haciendo pequeñas modificaciones a los pesos de las entradas para reducir las diferencias entre la salida obtenida y la salida deseada. El proceso de actualización de pesos es bastante simple. Como queremos obtener (o aproximarnos a) una salida deseada, tenemos que iterar sobre un conjunto de entrenamiento (de tamaño m) las veces necesarias, modificando los pesos de las conexiones, hasta minimizar el error en la salida. Si para el ejemplar de entrenamiento j , la salida es $Y(j)$ y la salida deseada es $Y_d(j)$, entonces la función de error está dada por:

$$e(j) = Y_d(j) - Y(j)$$

(Negnevitsky 2005, pág. 171)

Si $e(j)$ es positivo, se necesita incrementar $Y(j)$, pero si es negativo, se necesita disminuirlo. Tomando en cuenta que cada entrada contribuye $x_i(j) \times w_i(j)$ a la entrada total $X(j)$, nos damos cuenta que si el valor de entrada $x_i(j)$ es positivo, aumentar su peso $w_i(j)$ incrementa la salida del perceptrón, mientras que si $x_i(j)$ es negativa, un aumento en su peso $w_i(j)$ disminuye el valor de salida del perceptrón. Por lo tanto se puede establecer la siguiente regla de aprendizaje:

$$w_i(j+1) = w_i(j) + \alpha \times x_i(j) \times e(j)$$

(Negnevitsky 2005, pág. 171)

donde α es la **taza de aprendizaje**, una variable no negativa menor a 1.

Resumen del algoritmo de aprendizaje

En pocas palabras, se necesitan cuatro pasos para que un perceptrón aprenda a clasificar las entradas (Negnevitsky 2005, pág. 172):

1. **Inicialización:** Fijar los pesos iniciales w_1, w_2, \dots, w_n y el umbral θ a números aleatorios en el rango $[-0.5, 0.5]$ (recomendado).

2. **Activación:** Para un ejemplar j , activar el perceptrón aplicando las entradas $x_1(j), x_2(j), \dots, x_n(j)$.

$$Y(j) = f \left(\sum_{i=1}^n x_i(j)w_i(j) - \theta \right)$$

donde n es el número de entradas y f es la función de activación.

3. **Entrenamiento de pesos:** Se actualizan los pesos del perceptrón:

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

donde $\Delta w_i(t)$ es la corrección del peso al tiempo t para el ejemplar j , que se calcula de la siguiente forma:

$$\Delta w_i(t) = \alpha \times x_i(j) \times e(j)$$

El umbral θ se queda fijo en el valor que le fue asignado al inicio.

4. **Iteración:** Repetir los pasos 2 y 3 para cada ejemplar del conjunto de entrenamiento hasta minimizar lo mejor posible el error. Si el resultado no es satisfactorio, continuar ejecutando otra vez desde el primer ejemplar.

Desarrollo e implementación

Deberán crear dos perceptrones, uno que aprenda la operación AND y otro la operación OR, ambas de tres variables. Cada neurona tendrá cuatro entradas, tres para las entradas de la operación lógica y una para el sesgo θ .

x_1	x_2	x_3	Salida
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a) Operación AND

x_1	x_2	x_3	Salida
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(b) Operación OR

Su aplicación deberá permitir elegir distintos conjuntos de entrenamiento, en particular, tienen que especificar al menos 5 conjuntos de entrenamiento :

7. Perceptrón

1. $[[0,0,0,0], [1,1,1,1]]$ para AND y OR
2. El conjunto que contenga todas las combinaciones posibles de entradas
3. Y tres conjuntos más con elementos distintos, tomados de las tablas mostradas anteriormente.

Con el formato:

$$[x_1, x_2, x_3, \text{Salida}]$$

Esto para que se den una idea ustedes de lo importante que son los conjuntos de entrenamiento de un perceptrón. También tienen que mostrar el proceso de entrenamiento, y una vez entrenado el perceptrón, debe permitir hacer consultas de estas operaciones lógicas.

Para probar su perceptrón, deberán utilizar el siguiente conjunto de entradas:

$$[[0,0,0], [1,0,1], [0,0,1], [1,1,1]]$$

PD: Se les recomienda que no se compliquen y usen la función escalón (*step function*) como función de activación.

Requisitos y resultados

Entreguen el código donde implementaron el perceptrón y su entrenamiento. Generen un reporte con observaciones de cómo se comporta el perceptrón con cada uno de los conjuntos de entrenamiento. Lo podrán programar en Java o Python. No olviden comentar su código.

8 | Regresión

Víctor Germán Mijangos de la Cruz
Verónica Esther Arriola Ríos

Esta práctica introduce el concepto de *regresión logística* para estimar probabilidades de datos, así como para la clasificación de estos. Se definirá el algoritmo de aprendizaje para la regresión logística y se verán sus similitudes con el Perceptrón.

Objetivo

Que el alumno conozca el concepto de Regresión Logística, sus aplicaciones tanto para la estimación de distribuciones de probabilidad, como para su clasificación. Que el alumno implemente el algoritmo de aprendizaje para poder estimar las probabilidades en un problema de clasificación binaria.

Introducción

El aprendizaje supervisado comprende dos tipos de problemas básicos: 1) la clasificación; y 2) la regresión. Estos problemas pueden definirse a partir del tipo de valores que se esperan a la salida. En la clasificación, la salida de los métodos es una categoría, un valor discreto; por ejemplo $Y = \{0, 1, 2, \dots, N\}$. En el caso de la regresión, el valor de salida es continuo, ya sea \mathbb{R} o un intervalo.

Los modelos lineales determinan una función lineal para poder solucionar alguno de estos problemas. Una función lineal multivariable será una función de la forma:

$$a(x) = \sum_{i=1}^n x_i w_i + \theta$$

Los modelos lineales utilizan una función de este tipo para tratar los datos. Por ejemplo, el Perceptrón es un modelo lineal de clasificación, donde el valor final es determinado como 0, si $a(x)$ es un valor negativo, o 1 si es positivo.

8. Regresión

En el caso de la regresión logística lineal, lo que buscamos no es una clasificación, sino una regresión. En particular, la regresión logística obtendrá una probabilidad; es decir, un valor continuo en el intervalo $[0, 1]$. Ya que se trata de un modelo lineal, usará una función lineal para obtener el valor de regresión. A continuación desarrollamos con más detalle el modelo de regresión logística.

Regresión logística lineal

Se trata de un modelo de regresión para obtener valores probabilísticos en una distribución binaria. Así, dado un dato de entrada x , representado como un vector, se obtiene la probabilidad de que este vector pertenezca a la clase 1 o a la clase 0. En particular, nos enfocaremos en la probabilidad de la clase 1. Dado un conjunto de datos X y el conjunto de clases Y , asumiremos que la distribución sobre Y dado X es binaria o Bernoulli. En este sentido, buscamos estimar:

$$p := P(Y = 1|x)$$

La probabilidad de la clase 0 se puede obtener de esta como $P(Y = 0|x) = 1 - P(Y = 1|x) = 1 - p$. La función logística, que es la función a partir de la cual podemos obtener p , se obtiene asumiendo que el logit de la distribución es lineal. El *logit* es una estadística que determina el logaritmo de la verosimilitud de la distribución.

El logit es igual a 0 cuando $p = 1 - p$, es decir, cuando el evento es aleatorio, toma valores positivos cuando $p > 1 - p$ y valores negativos cuando $p < 1 - p$. Así la función logit está determinada como $\ln \frac{p}{1-p}$. Finalmente, si asumimos que el logit es lineal, tenemos:

$$\ln \frac{p}{1-p} = \sum_{i=1}^n x_i + \theta$$

Al despejar p en esta función obtenemos la llamada función logística:

$$p = \sigma(x) = \frac{1}{1 + e^{-(\sum_{i=1}^n x_i + \theta)}}$$

La función logística toma valores entre 0 y 1, y toma el valor 0.5 cuando $\sum_{i=1}^n x_i + \theta = 0$. En la Figura 8.1. Los valores de 0 y 1 sólo se alcanzan en el límite. Los valores w_i , $i = 1, \dots, n$ y θ son los parámetros que se deben aprender. La función logística es entonces la función que determina el modelo de regresión logística.

Definición 8.1: Regresión logística

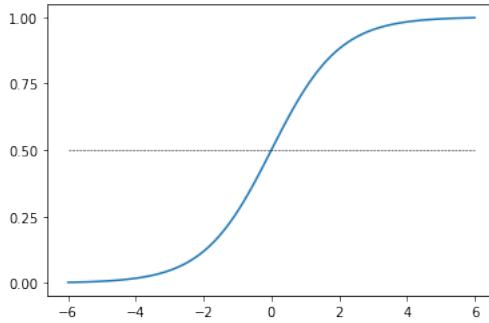


Figura 8.1 Función logística

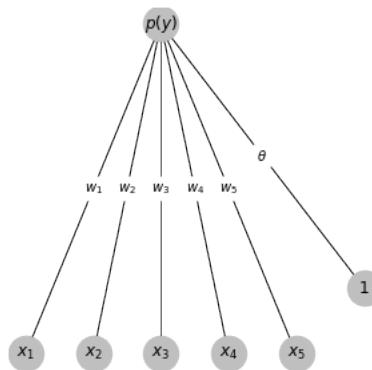


Figura 8.2 Modelo gráfico del modelo de regresión lineal

La regresión logística es un modelo gráfico lineal para estimar probabilidades en distribuciones binarias. En este modelo, la distribución probabilística está definida por la función logística:

$$\sigma(x; w, \theta) = \frac{1}{1 + e^{-(\sum_{i=1}^n x_i + \theta)}}$$

Donde $w = (w_1 \ w_2 \ \dots \ w_n)$ y θ son los parámetros del modelo que debemos aprender.

El modelo gráfico que define la regresión logística se puede ver en la Figura 8.2. Como se puede observar, la gráfica del modelo es similar a la de un modelo de Bayes Ingenuo; sin embargo, ambos modelos operan de forma diferente. Se dice que el modelo de Bayes Ingenuo es un modelo generativo, mientras que el modelo de regresión logística es un modelo discriminativo.

Estimación del modelo

Para aprender los parámetros de la regresión logística, utilizaremos el algoritmo de descenso por el gradiente. Básicamente, el algoritmo de descenso por el gradiente busca encontrar el mínimo de una función objetivo a partir de observar el comportamiento del vector gradiente (si la dirección es ascendente, nos indica que nos alejamos del mínimo, por lo que debemos reducir los valores de los parámetros; por el contrario, si la dirección del gradiente es descendente, seguimos avanzando por esta dirección para alcanzar el mínimo). Este algoritmo se define como:

$$x \leftarrow x - \alpha \nabla_x f(x)$$

Es decir, actualizamos los valores de los parámetros restando el gradiente ponderado por un valor real α . Al valor α se le conoce como taza de aprendizaje y funciona de manera idéntica que en el Perceptrón.

En el caso de la regresión logística, la función objetivo (ya que trabajamos con probabilidades) será la estimación por máxima verosimilitud logarítmica. Pero esta función busca maximizar los valores por lo que tomaremos el negativo de esta función.¹ Esta función está dada como:

$$J(w, \theta) = -y \ln(\sigma(x; w, \theta)) - (1 - y) \ln(1 - \sigma(x; w, \theta))$$

Dado que tenemos una distribución binaria, se toman dos términos: el término $y \ln(\sigma(x; w, \theta))$ que corresponde a la probabilidad de la clase 1, y el término $(1 - y) \ln(1 - \sigma(x; w, \theta))$ que corresponde a la probabilidad de la clase 0. Claramente si el dato x de entrada pertenece a la clase 1, minimizar el primer término corresponde a maximizar la probabilidad $\sigma(x; w, \theta)$. Análogamente, se maximiza la probabilidad de la clase 0 cuando los datos de entrada son de esta clase.

Para aplicar el descenso por el gradiente debemos obtener el vector gradiente, lo que implica obtener las derivadas parciales de la función objetivo con respecto a los parámetros entrenables. Podemos observar que:

$$\begin{aligned} \frac{\partial J}{\partial w_i} &= -y \frac{\partial}{\partial w_i} \ln(\sigma(x; w, \theta)) - (1 - y) \frac{\partial}{\partial w_i} \ln(1 - \sigma(x; w, \theta)) \\ &= (\sigma(x; w, \theta) - y)x_i \end{aligned}$$

Pues la derivada de la sigmoide es $\sigma(x; w, \theta)(1 - \sigma(x; w, \theta))$ y desarrollando por los dos casos, $y = 1$ y $y = 0$, podemos obtener el resultado. De igual forma es fácil notar que para θ tenemos:

¹A la función objetivo definida de esta forma también se le conoce como entropía cruzada

$$\begin{aligned}\frac{\partial J}{\partial \theta} &= -y \frac{\partial}{\partial \theta} \ln(\sigma(x; w, \theta)) - (1-y) \frac{\partial}{\partial \theta} \ln(1 - \sigma(x; w, \theta)) \\ &= (\sigma(x; w, \theta) - y)\end{aligned}$$

En el Algoritmo 3 se describe la implementación del aprendizaje en la regresión logística.

Algoritmo 3 Algoritmo de aprendizaje para la regresión logística

```

function FIT-LINEAR-REGRESSION( $X, Y, \eta, T$ )
   $w, \theta \leftarrow \text{RANDOM}(w, \theta)$ 
  for  $t \leftarrow 1$  hasta  $T$  do
    for  $x, y \in X, Y$  do
       $f(x) \leftarrow \sigma(x; w, \theta)$ 
       $w \leftarrow w - \eta(f(x) - y)x$ 
       $\theta \leftarrow \theta - \eta(f(x) - y)$ 
    end for
  end for
  return La función  $\sigma(\cdot; w, \theta)$  con parámetros óptimos
end function
```

El algoritmo para aprendizaje en regresión logística debe tomar un conjunto de datos de entrenamiento X y las clases cuyas probabilidades se desean estimar Y . Además se indica el rango de aprendizaje η y el número máximo de iteraciones T . Los valores de w y θ se inician aleatoriamente y se ajustan hasta obtener una solución adecuada.

Clasificación con regresión logística

El modelo de regresión logística permite obtener la probabilidad de una clase dado un vector de entrada. Con esta probabilidad se puede realizar una clasificación, pues sabemos que la probabilidad de la clase 1 está dada por la función logística, mientras que la probabilidad de la clase 0 podrá obtener fácilmente de esta. Por tanto, podemos definir una función de clasificación a partir de la regresión logística como sigue:

$$\hat{y} = \begin{cases} 1 & \text{si } \sigma(x; w, \theta) \geq 0.5 \\ 0 & \text{si } \sigma(x; w, \theta) < 0.5 \end{cases}$$

Esta función es similar a la función $\arg \max_y P(Y = y|x)$ pues toma como clase aquella cuya probabilidad sea la más alta, pues si la clase 0 es más alta, dado que $P(Y = 0|x) = 1 - \sigma(x; w, \theta)$, entonces necesariamente la clase 1 (cuya probabilidad es $P(Y = 1|x) = \sigma(x; w, \theta)$) será más baja; y viceversa. También podemos notar que, como señalamos anteriormente $\sigma(x; w, \theta) = 0.5$ cuando $\sum_{i=1}^n w_i x_i + \theta = 0$ (véase Figura 8.1),

8. Regresión

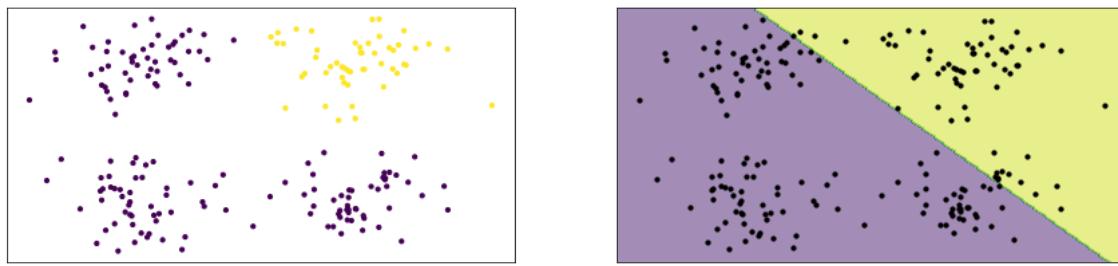


Figura 8.3 A la izquierda los datos de entrenamiento, a la derecha las regiones de clasificación que se obtienen al entrenar el modelo.

la clasificación corresponde a la función escalón del Perceptrón; es decir, a la función dada por:

$$\hat{y} = \begin{cases} 1 & \text{si } \sum_{i=1}^n w_i x_i + \theta \geq 0 \\ 0 & \text{si } \sum_{i=1}^n w_i x_i + \theta < 0 \end{cases}$$

En conclusión, podemos ver que la regresión logística está ampliamente relacionada con el Perceptrón. En la Figura 8.3 se puede ver un conjunto de datos para entrenar el modelo a la izquierda, y las regiones de clasificación que aprende el modelo a la derecha. Claramente, por lo que acabamos de señalar de su relación con el Perceptrón, la clasificación realizada por el modelo de regresión será lineal.

Desarrollo

En esta práctica se implementará el modelo de regresión lineal y el algoritmo de aprendizaje para esta en un conjunto de datos lógico. Se propone generar un conjunto de datos en base al problema lógico AND, pero introduciendo ruido para dispersar los datos en el espacio \mathbb{R}^2 . Se entrenará un modelo de regresión y se obtendrán las probabilidades de que los datos pertenezcan a una clase o a otra.

Implementación

Para la implementación del código se manejarán arreglos en numpy y se definirá una clase para el modelo de regresión lineal. Esta clase deberá tener las funciones `fit`, `predict_proba` y `predict`. Se deberán seguir los siguientes pasos:

- Se generarán 200 ejemplos de entrenamiento definidos por la función AND más un ruido. La función AND está dada por la tabla de verdad:

x_1	x_2	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Se generarán 150 datos para la clase 0 y 50 datos para la clase 1. Para que estos datos sean variados, se sumará ruido gaussiano a estos datos. Se sugiere generar el ruido de la siguiente forma:

```
numpy.random.normal(loc=0,scale=1,size=(200,2))
```

Esto generará ruido con media 0 y varianza 1 para los 200 datos de dos dimensiones. Se sumará a los este ruido escalado por un factor; se sugiere escalar el ruido por un factor entre 0.1 y 0.2.

- Se generará la clase LogisticRegression. Dentro de esta clase, se definirán las funciones `fit`, `predict_proba` y `predict`.
- La función `fit` deberá tomar como argumentos los datos X y sus clases Y para el entrenamiento; además se deberá indicar el rango de aprendizaje (se sugiere un valor de 0.1) y un número máximo de iteraciones (se sugiere 100). Esta función aprenderá el algoritmo de aprendizaje (Algoritmo 3). Se sugiere inicializar los parámetros de manera aleatoria con valores en el intervalo $[-0.5, 0.5]$.
- La función `predict_proba` tomará como argumento un vector o varios vectores de entrada x y devolverá la probabilidad de clase de estas entradas. Puede utilizarse esta función dentro de la función de entrenamiento `fit`.
- La función `predict` tomará como argumento un vector o varios vectores de entrada x y devolverá la clase propuesta para estas entradas. Es decir, esta función se encargará de clasificar los datos de entrada.
- Se sugiere comparar los resultados de clasificación con aquellos obtenidos con el Perceptrón.

Requisitos y resultados

Se tendrá que entregar la clase `LogisticRegression` adecuadamente programada y documentada. Como se ha mencionado, la clase deberá contener las funciones `fit`, `predict_proba` y `predict`, todas estas adecuadamente realizadas y funcionando de la manera indicada. Además, se usará esta clase para entrenar y obtener las clases de los datos que ya se han indicado.

9 | Factores

Luis Alfredo Lizárraga Santos
Verónica Esther Arriola Ríos

Objetivo

Que el alumno implemente una clase Factor para familiarizarse con las operaciones entre factores que se utilizan para realizar inferencia en redes Bayesiana.

Introducción

Un factor es una estructura matemática que nos ayudará a sistematizar las diversas operaciones requeridas para operar con distribuciones de probabilidad discretas. Por ello, en esta práctica, el alumno se familiarizará con estas estructuras e implementará sus operaciones en un lenguaje de programación. Posteriormente, la biblioteca creada será la herramienta con la cual se resolverán problemas de aplicación diversos.

Definición 9.1: Factor

"Sea D un conjunto de variables aleatorias [y $\text{Val}(D)$ una asignación de valores a estas variables]. Se define un Factor ϕ como una función de $\text{Val}(D)$ a \mathbb{R} . Un factor es no-negativo si todas sus entradas son no negativas. Al conjunto de variables D se le llama *alcance del factor* y se denota como $\text{Alcance}(\phi)$ "

(Koller y Friedman 2009, pág. 104).

Tomando como base la definición de Koller y Friedman, podemos realizar un parafraseo y afirmar que un factor es una estructura matemática definida sobre un conjunto de variables donde cada variable puede tomar valores de su dominio, el cual debe ser finito. El factor asocia un número real a cada posible asignación de valores de esas variables.

A	$\phi(A)$
0	.3
1	.7

(a) Factor A

B	$\phi(B)$
0	.6
1	.4

(b) Factor B

C	$\phi(C)$
0	.2
1	.8

(c) Factor C

Figura 9.1 Factores sobre variables aleatorias binarias.

En cuanto a las operaciones de factores que se explicarán en esta práctica, nos basamos en las operaciones que presentan Koller y Friedman: multiplicación, reducción, marginalización y normalización.

Utilizaremos a los factores para realizar operaciones con distribuciones de probabilidad, sin embargo es importante notar que, debido a la misma definición de factor, el resultado de cada operación entre factores no siempre será una medida de probabilidad, pues esta limita su valores al intervalo $[0, 1]$, mientras que los factores ocupan todo \mathbb{R} . Adicionalmente, sus operaciones no siempre incluyen las normalizaciones requeridas por las distribuciones de probabilidad; pero esto se convertirá en una ventaja para nosotros, como veremos más adelante.

Operaciones entre Factores

Multiplicación

Definición 9.2: Multiplicación

“Sean X , Y y Z tres conjuntos disjuntos de variables, y sean $\phi_1(X, Y)$ y $\phi_2(Y, Z)$ dos factores. Se define el factor producto $\phi_1 \times \phi_2$ como un factor $\psi : \text{Val}(X, Y, Z) \mapsto \mathbb{R}$ de la forma siguiente:

$$\psi(X, Y, Z) = \phi_1(X, Y) \cdot \phi_2(Y, Z)$$

”

(Koller y Friedman 2009, pág. 107).

La operación de multiplicación es sencilla. Si los conjuntos de variables de los factores a multiplicar no tienen elementos en común, se multiplica cada entrada del factor A por cada entrada del Factor B. El alcance del factor resultante es la unión de los alcances de los factores a multiplicar (Koller y Friedman 2009, pág. 107).

Por ejemplo: Tenemos los tres factores A, B y C de la Figura 9.1. Para obtener el factor $AB = A \times B$, se multiplicaría el renglón $A = 0$ con $B = 0$, luego $A = 0$ con $B = 1$, $A = 1$ con $B = 0$ y por último $A = 1$ con $B = 1$ como en la Figura 9.2.

9. Factores

A	B	$\phi(A, B)$
0	0	$(.3) * (.6) = 0.18$
0	1	$(.3) * (.4) = 0.12$
1	0	$(.7) * (.6) = 0.42$
1	1	$(.7) * (.4) = 0.28$

Figura 9.2 Factor AB

Si el alcance de ambos factores tiene variables en común, se debe asegurar que tengan el mismo valor en cada renglón por multiplicar en ambos factores. Por ejemplo, si se tienen los factores AB y AC, al momento de multiplicar el renglón $\{A = 0, B = 0\}$ se debe seleccionar los renglones donde $A = 0$ en el factor AC, estos son $\{A = 0, C = 0\}$ y $\{A = 0, C = 1\}$, como muestran Figura 9.3 y Figura 9.4.

A	B	$\phi(A, B)$
0	0	$(.3) * (.6) = 0.18$
0	1	$(.3) * (.4) = 0.12$
1	0	$(.7) * (.6) = 0.42$
1	1	$(.7) * (.4) = 0.28$

(a) Factor $AB = A \times B$

A	C	$\phi(A, C)$
0	0	$(.3) * (.2) = 0.6$
0	1	$(.3) * (.8) = 0.24$
1	0	$(.7) * (.2) = 0.14$
1	1	$(.7) * (.8) = 0.56$

(b) Factor $AC = A \times C$ **Figura 9.3** Multiplicandos.

A	B	C	$\phi(A, B, C)$
0	0	0	$[(.3) * (.6)] * [(.3) * (.2)] = 0.0108$
0	0	1	$[(.3) * (.6)] * [(.3) * (.8)] = 0.0432$
0	1	0	$[(.3) * (.4)] * [(.3) * (.2)] = 0.0072$
0	1	1	$[(.3) * (.4)] * [(.3) * (.8)] = 0.0288$
1	0	0	$[(.7) * (.6)] * [(.7) * (.2)] = 0.0588$
1	0	1	$[(.7) * (.6)] * [(.7) * (.8)] = 0.2352$
1	1	0	$[(.7) * (.4)] * [(.7) * (.2)] = 0.0392$
1	1	1	$[(.7) * (.4)] * [(.7) * (.8)] = 0.1568$

Figura 9.4 Factor $AB \times AC$

Es importante hacer notar que al multiplicar los factores AB y AC no se estaría obteniendo la probabilidad conjunta de A, B y C, si no alguna otra función $\phi(A, B, C)$.

Reducción

Definición 9.3: Reducción

“Sea $\phi(Y)$ un factor y $U = u$ una asignación para $U \subseteq Y$. Se define la reducción de un factor ϕ al contexto $U = u$, denotado como $\phi[U = u]$ (y abreviado como $\phi[u]$), como un factor con alcance $Y' = Y - U$ de tal forma que

$$\phi[u](y') = \phi(y', u)$$

”

(Koller y Friedman 2009, pág. 111).

La operación de reducción consiste en seleccionar un valor de alguna variable del factor y sólo tomar los renglones que cumplen con el valor dado de la variable. Por ejemplo: Se tiene el factor AB de la Figura 9.5.

A	B	$\phi(A, B)$
0	0	.18
0	1	.12
1	0	.42
1	1	.28

Figura 9.5 Factor AB

Si se desea reducir con $A = 0$, el resultado es el factor en la Figura 9.6.

B	$\phi(B)$
0	.18
1	.12

Figura 9.6 Factor B

Normalización

Para normalizar un factor, esto es, que los valores que toma el factor sumen 1, basta con sumar todos los valores asociados a las asignaciones y dividir cada uno entre esta suma. Por ejemplo: Tenemos el factor $\phi(B)$ obtenido en la Figura 9.6, la suma de sus posibles valores es .3, entonces el factor normalizado queda como en la Figura 9.7.

9. Factores

A	B	$\phi(B)$
0	0	(.18/.3)= .6
0	1	(.12/.3)= .4

Figura 9.7 Factor B normalizado**Marginalización**

“Sea X un conjunto de variables y sea $Y \notin X$ una variable y $\phi(X, Y)$ un factor. Se define la marginalización de Y en ϕ , denotado como $\sum_Y \phi$, como un factor ψ sobre X de tal forma que

$$\psi(X) = \sum_Y \phi(X, Y)$$

” (Koller y Friedman 2009, pág. 297).

La operación de marginalización consiste en tomar la variable a marginalizar, sumar los valores en los renglones en que cambia su valor pero el de las demás variables no, y asignar esta suma al renglón correspondiente de las variables restantes. Por ejemplo: tenemos el factor AB y deseamos marginalizar la variable B. Entonces, tomamos los renglones donde A=0 y los sumamos [Figura 9.8], tomamos los renglones donde A=1 y los sumamos [Figura 9.9].

A	B	$\phi(A, B)$
→ 0	0	.18
→ 0	1	.12
1	0	.42
1	1	.28

(a) Factor AB
(b) Factor A

Figura 9.8 Paso 1

A	B	$\phi(A, B)$
0	0	.18
0	1	.12
→ 1	0	.42
→ 1	1	.28

(a) Factor AB
(b) Factor A

Figura 9.9 Paso 2

Desarrollo e implementación

La práctica consiste en crear una clase Factor que implemente las operaciones de multiplicación, reducción y normalización de factores y marginalización de variables. Todo esto utilizando el lenguaje de programación Python. Para guiar tu implementación, el código auxiliar de esta práctica consiste en un script de uso llamado PruebaFactores.py, que deberá funcionar correctamente utilizando tu clase, la cual deberá estar implementada en un archivo Factores.py al lado de éste.

Implementación

1. Crear una clase Variable, con los siguientes atributos: nombre y valores_posibles y sobrescribir el método `__str__`.
2. Crear una clase Factor que contenga los atributos:
 - alcance: una lista de objetos de clase Variable.
 - valores: una lista de valores asociados a cada renglón.
3. Programar un método auxiliar `_generar_tabla_de_valores`, que sólo ejecutará su algoritmo la primera vez que sea invocado y su uso primordialmente será para poder imprimir en pantalla el contenido del factor, es decir, servirá para sobreescrbir el método `__str__` de la clase Factor.

Este método deberá generar, de forma explícita, la tabla de asignaciones posibles a las variables en el alcance del factor. Estas combinaciones serán listadas en formato semejante a un reloj digital, de modo que la última variable en el atributo alcance sea la que varíe su valor más rápido. Los valores posibles se irán listando en el orden en que aparecen en el atributo `valores_posibles` del objeto de tipo Variable.

Observa que esta implementación no exige que los valores de las variables sean números, como indicaría la definición formal, pues en cualquier modo esto no es requisito para el correcto funcionamiento de las operaciones. Esta generalización podrá ser utilizada para visualizar resultados más rápidos de interpretar.

Por ejemplo, para las variables:

- Letras = ['a', 'b']
- Útiles = ['cuaderno', 'lápiz', 'goma']
- Números = ['2', '8']

La tabla de valores posibles se extiende como en la Figura 9.10.

Usando esto, sobreescrbir el método `__str__`.

9. Factores

Índice	Letras	Útiles	Números
0	a	cuaderno	2
1	a	cuaderno	8
2	a	lápiz	2
3	a	lápiz	8
4	a	goma	2
5	a	goma	8
6	b	cuaderno	2
7	b	cuaderno	8
8	b	lápiz	2
9	b	lápiz	8
10	b	goma	2
11	b	goma	8

Figura 9.10 Tabla de valores

TIP: Como tu método debe funcionar para cualquier número de variables en el alcance generar esta tabla requerirá una forma creativa de anidar ciclos `for`. En algún lugar necesitarás iterar sobre la misma tabla de valores que estás generando.

4. Programar un método auxiliar que reciba como parámetro un diccionario, cuyas llaves sean variables y los valores sean el valor asignado a cada una. Este método deberá obtener el índice en la tabla de valores que corresponda a esta asignación. Observa que para obtener el índice no es necesario haber desarrollado la tabla explícitamente, se utiliza un polinomio de direccionamiento.

Por ejemplo, el polinomio de direccionamiento para un factor de 3 variables (A, B y C) se vería así:

$$\text{índice} = (\text{pos}(a) * |\text{B}| * |\text{C}|) + (\text{pos}(b) * |\text{C}|) + \text{pos}(c)$$

donde $\text{pos}(a)$ es la posición del valor asociado a la variable A en su lista de valores posibles y $|\text{A}|$ es el tamaño de esta lista.

En su forma más general el polinomio de direccionamiento tiene la forma:

$$\text{índice} = \sum_{\text{Var} \in D} \left(\text{pos}(\text{Val}(\text{Var})) \prod_{\text{Var} \in D} |\text{Var}| \right)$$

donde $\text{Var} \in D$ son todas las variables a la derecha de Var en la lista de `valores_posibles`.

TIP: Pueden factorizar los términos comunes (los tamaños de las listas) e implementar una función auxiliar que calcule recursivamente el valor del polinomio de direccionamiento.

5. Implementa las operaciones: multiplicación, reducción, normalización y marginalización.

- En el caso de la multiplicación, considera también el caso de multiplicar por un escalar. Si el parámetro recibido es un escalar, el factor resultado tendrá el mismo alcance y sus valores serán multiplicados todos por el escalar.
- Para la marginalización, si se pide marginalizar la única variable del factor, suma los renglones y devuelve el escalar.
- En la reducción, si se reduce la única variable, devuelve el valor del factor que corresponda al valor indicado de la variable.

TIP: Crea primero el factor resultado con lista de variables en su alcance. Para cada renglón en la tabla de valores de este factor, encuentra los renglones relevantes en los operandos utilizando el método auxiliar mencionado en el punto anterior y realiza la operación correspondiente.

Esta práctica deberá ser implementada usando Python.

Requisitos y resultados

Deberán hacer casos de prueba para marginalización, reducción, normalización y multiplicación de factores. Basta con incluir un pequeño *script* con sus casos de prueba. No es necesario que lo hagan a prueba de todo, pero sí que funcionen cuando se cumplen los requisitos para cada operación.

Para evaluar y calificar la práctica es necesario que se implementen todos los métodos mencionados e indicados, respetando las especificaciones de estilo y documentación del lenguaje de programación que usarán. Es completamente válido utilizar bibliotecas adicionales si lo consideran necesario, así como la creación y uso de sus propios métodos auxiliares si lo desean.

En la figura 9.11 pueden apreciar el resultado de crear 3 factores y ejecutar las operaciones de multiplicación, normalización, reducción y marginalización entre ellos.

9. Factores

(a) Creación de factores f1 y f2

```
python3 factores.py
Factor 1
Variables:
-----
Nombre: A | valores: { 1, 2 }
-----
A | Prob
1 | 1
2 | 2
Factor 2
Variables:
-----
Nombre: B | valores: { 5, 10, 15 }
Nombre: C | valores: { 1, 2, 3 }
-----
B | Prob
5 1 | 5
5 2 | 10
5 3 | 15
10 1 | 10
10 2 | 20
10 3 | 30
15 1 | 15
15 2 | 30
15 3 | 45
```

(b) Multiplicando f1 y f2, para obtener f3

```
Multiplicando f1 y f2: f3
Variables:
-----
Nombre: A | valores: { 1, 2 }
Nombre: B | valores: { 5, 10, 15 }
Nombre: C | valores: { 1, 2, 3 }
-----
A B C | Prob
1 5 1 | 0.0093
1 5 2 | 0.0185
1 5 3 | 0.0278
1 10 1 | 0.0185
1 10 2 | 0.037
1 10 3 | 0.0556
1 15 1 | 0.0278
1 15 2 | 0.0556
1 15 3 | 0.0833
2 5 1 | 0.0185
2 5 2 | 0.037
2 5 3 | 0.0556
2 10 1 | 0.037
2 10 2 | 0.0741
2 10 3 | 0.1111
2 15 1 | 0.0556
2 15 2 | 0.1111
2 15 3 | 0.1667
```

(c) Normalizando f3

```
Normalizando el resultado de la multiplicación
Variables:
-----
Nombre: A | valores: { 1, 2 }
Nombre: B | valores: { 5, 10, 15 }
Nombre: C | valores: { 1, 2, 3 }
-----
A B C | Prob
1 5 1 | 0.0093
1 5 2 | 0.0185
1 5 3 | 0.0278
1 10 1 | 0.0185
1 10 2 | 0.037
1 10 3 | 0.0556
1 15 1 | 0.0278
1 15 2 | 0.0556
1 15 3 | 0.0833
2 5 1 | 0.0185
2 5 2 | 0.037
2 5 3 | 0.0556
2 10 1 | 0.037
2 10 2 | 0.0741
2 10 3 | 0.1111
2 15 1 | 0.0556
2 15 2 | 0.1111
2 15 3 | 0.1667
```

(d) Reduciendo y marginalizando

```
Reduciendo la variable A con valor 1 de f3
Variables:
-----
Nombre: B | valores: { 5, 10, 15 }
Nombre: C | valores: { 1, 2, 3 }
-----
B | Prob
5 1 | 0.0093
5 2 | 0.0185
5 3 | 0.0278
10 1 | 0.0185
10 2 | 0.037
10 3 | 0.0556
15 1 | 0.0278
15 2 | 0.0556
15 3 | 0.0833
Marginalizando la variable B de f3
Variables:
-----
Nombre: A | valores: { 1, 2 }
Nombre: C | valores: { 1, 2, 3 }
-----
A C | Prob
1 1 | 0.0556
1 2 | 0.1110999999999999
1 3 | 0.1667000000000001
2 1 | 0.1110999999999999
2 2 | 0.2222
```

Figura 9.11 Ejecución del código solución: crea dos factores, los multiplica, normaliza, reduce y marginaliza

No olviden documentar y comentar su código.

10 | Inferencias en Redes de Bayes usando factores

Verónica Esther Arriola Ríos

Esta práctica es un pequeño complemento a la práctica *Factores*.

Objetivo

El alumno utilizará la clase factor para resolver consultas a una red de Bayes.

- Que el alumno identifique las ventajas de utilizar factores para calcular distribuciones de probabilidad sobre realizar los cálculos en la forma tradicional: con sumas y productos, considerando cada valor posible manualmente.
- Identificar la relación entre marginalización en probabilidad y marginalización de factores.
- Combinar las operaciones reducción y normalización de factores para calcular la reducción en distribuciones de probabilidad.
- Que alumno pueda mapear de una ecuación con sumas y productos de distribuciones a las operaciones correspondientes con factores.

Introducción

Para realizar esta práctica se tomarán ejemplos del problema de la fiesta de Pedro, por lo que necesitarás la gráfica obtenida y las distribuciones de probabilidad completadas en teoría. [Figura 10.1]

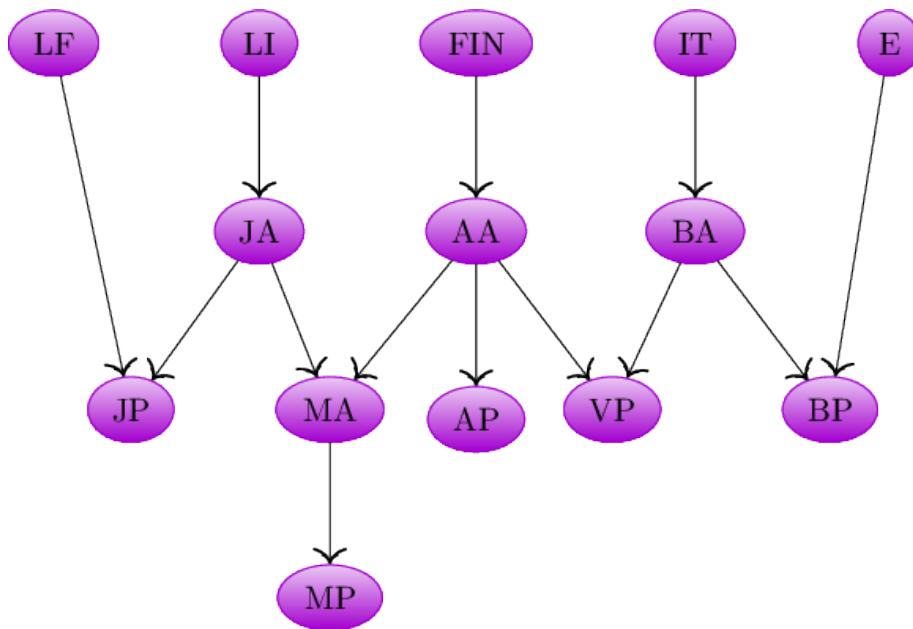


Figura 10.1 Gráfica con las relaciones de dependencia condicional del problema de la Fiesta de Pedro. El significado de los nodos y las distribuciones de probabilidad correspondientes se pueden encontrar entre los textos del curso.

Desarrollo e implementación

Para resolver los ejercicios siguientes deberás utilizar la clase que programaste en la práctica Factores, por lo que será necesario que la entregues empaquetada en esta práctica también; si deseas hacer correcciones es un buen momento para incluirlas.

Requisitos y resultados

Para las instrucciones siguientes asumiremos que el código está en Python. Si alguien decidió trabajar en otro lenguaje deberá consultar directamente con el ayudante del curso y adaptar las indicaciones a las convenciones del lenguaje elegido.

1. Para esta práctica deberás entregar dos archivos:
 - a) Factor.py. Es el código de la práctica anterior y será necesario para ejecutar esta.
 - b) PedroFactor.py. En este *script* colocarás las funciones para resolver los problemas indicados. Cada función que programes deberá:
 - 1) Imprimir al inicio cuál es la *consulta* (probabilidad) que está realizando.

- 2) Los factores con resultados intermedios relevantes. Antes de imprimir el factor, imprime su significado. Por ejemplo: $P(\text{ma}|\text{AP}, \text{VP})$, $P(\text{MA}, \text{mp})$ ó $P(\text{MA}, \text{JA})P(\text{LI}|bp)$.
- 3) El resultado final, igualmente con algún formato que lo haga resaltar de las demás operaciones.

Al final, en la sección `if __name__ == '__main__':` mandar ejecutar todas las funciones mostrando tus resultados para cada ejercicio.

Notación: Para abreviar sumas/marginalizaciones sucesivas, se utilizará la siguiente abreviatura:

$$\sum_{A,B,C} = \sum_A \sum_B \sum_C f(A, B, C)$$

Las tareas que debes realizar son las siguientes:

1. Crea todas las variables correspondientes a las variables aleatorias de la fiesta de Pedro como variables globales del *script*.
2. Crea todos los factores correspondientes a las distribuciones de probabilidad, también como variables globales.
3. Crea una función por cada ecuación, donde resuelvas directamente la traducción de las operaciones siguientes, en el orden en el que están especificadas. Para cada versión imprime el número de renglones que tienen tus factores después de cada multiplicación y marginalización. Observa que, al final, todas deben dar el mismo resultado

$$P(VP, BP) = \sum_{AA, BA, FIN, IT, E} P(VP|AA, BA)P(BP|BA, E)P(AA|FIN) \\ P(BA|IT)P(FIN)P(IT)P(E) \quad (10.1)$$

$$P(VP, BP) = \sum_{AA, BA} P(VP|AA, BA) \sum_E P(BP|BA, E) \sum_{FIN} P(AA|FIN) \\ \sum_{IT} P(BA|IT)P(FIN)P(IT)P(E) \quad (10.2)$$

$$P(VP, BP) = \sum_{AA, BA} \left\{ P(VP|AA, BA) \left[\sum_{FIN} P(AA|FIN)P(FIN) \right] \right\} \\ \left[\sum_E P(BP|BA, E)P(E) \right] \left[\sum_{IT} P(BA|IT)P(IT) \right] \quad (10.3)$$

4. Calcula la probabilidad de que María, Alicia y Víctor estén en la fiesta $P(\text{MP}, \text{AP}, \text{VP})$.

10. Inferencia

5. Calcula la probabilidad de que haya llovido el día que enviaron la invitación dado que María y Alicia estuvieron presentes. Para ello seguirás el método siguiente. Recuerda que, por definición de probabilidad condicional:

$$P(LI|mp, ap) = \frac{P(LI, mp, ap)}{P(mp, ap)}$$

- a) Calcula la ecuación correspondiente para obtener $P(LI, mp, ap)$ usando la regla de la cadena para redes de Bayes. Factorízala según sea conveniente y escribe tu resultado en los comentarios de la función de python para este ejercicio.
- b) Utiliza tus factores para obtener el resultado correspondiente, pero antes de multiplicar y marginalizar, manda llamar la operación `reducir` sobre todos los factores que contengan a MP y AP, de modo que sólo te quedes con los renglones donde MP = mp y AP = ap.
- c) Normaliza tu resultado final, esta es la probabilidad que estabas buscando.

11 | Clasificador Bayesiano Ingenuo y aplicación de la clase Factor

Luis Alfredo Lizárraga Santos

Objetivo

Que el alumno utilice probabilidad en una aplicación de la vida real: modelar las probabilidades de que dado un texto, se pueda determinar a qué etiqueta pertenece: Ciencia, Tecnología, Cultura, etc.

Introducción

Tener la posibilidad de etiquetar un texto automáticamente permite que las empresas del ramo periodístico puedan hacer un mejor uso de sus activos humanos para generar notas de valor y no estar etiquetando manualmente cada nota que entregan sus correspondientes. Por otro lado, estas etiquetas permiten una búsqueda de noticias más acertada para el lector, ya que son palabras clave derivadas del texto de la noticia.

En esta práctica utilizaremos un algoritmo de Aprendizaje automático: el Clasificador Bayesiano Ingenuo (*Naive Bayes Classifier*).

Clasificador Bayesiano Ingenuo

Una manera sencilla de obtener una etiqueta para cualquier texto es utilizar un Clasificador Bayesiano Ingenuo, el cual se basa en el uso de Probabilidad y el Teorema de Bayes para predecir la etiqueta para un texto dado, obteniendo la probabilidad de cada etiqueta y retornando la más probable. A continuación veremos un ejemplo de cómo aplicar este clasificador, basado en una entrada del blog de Monkey Learn (Stecanella 2017)

Utilizando un Clasificador Bayesiano

Supongamos que queremos construir un clasificador que nos diga las etiquetas más probables para un texto y tenemos los siguientes datos de entrenamiento:

Texto	Etiqueta
de acuerdo con el portal de datos abiertos de la ciudad de méxico	CDMX
el gobierno capitalino trabaja en la seguridad en ciudad de méxico	CDMX
claudia sheinbaum aseguró que no se elevarán las tarifas para los capitalinos	CDMX
clasificación a octavos de final de la champions league	Deportes
la fifa determinó que deberá pagar seis millones de dólares	Deportes
con la oportunidad de clasificar a octavos, nápoles recibirá al rb salzburgo	Deportes
el presidente lamentó lo sucedido al alcalde de valle de chalco	Nacional
la comisión temporal de presupuesto del instituto nacional electoral	Nacional
por la mañana, durante su conferencia de prensa, el presidente	Nacional

Figura 11.1 Datos de entrenamiento

¿Cómo podríamos saber a qué etiqueta pertenece el texto “desde el aeropuerto capitalino fue trasladada”?

Por ser un clasificador Bayesiano, calcularemos la probabilidad de que el texto anterior tenga alguna de las etiquetas Deportes, Nacional o CDMX y tomaremos la más alta. Entonces, lo que buscamos obtener es

$$P(\text{CDMX}|x), P(\text{Nacional}|x), P(\text{Deportes}|x)$$

donde

$$x = \text{desde el aeropuerto capitalino fue trasladada}$$

Propiedades

Lo primero que debemos de hacer al crear un modelo de Aprendizaje Automático es definir qué propiedades se utilizarán para que nuestro algoritmo pueda determinar la etiqueta correcta. Una propiedad es un pedazo de información que se le da al algoritmo para que aprenda las correlaciones o patrones existentes. Un ejemplo de propiedades que se podrían utilizar si se quisiera saber si una persona es más propensa a consumir cervezas artesanales son: edad, sexo, nivel de ingreso, etc. y se excluirían datos que pueden no ser relevantes para el modelo como nombre, correo o si están casados.

Estas propiedades deben tener una representación numérica para que el algoritmo pueda entenderlas, pero en nuestro caso sólo tenemos texto. Para poder hacer cualquier tipo de cálculo primero debemos transformar el texto a una representación numérica,

esto lo logramos utilizando frecuencias de palabras, nos fijamos en cuántas veces aparece cada palabra por etiqueta.

Teorema de Bayes

Para lograr calcular probabilidades sobre frecuencias de palabras, utilizaremos el Teorema de Bayes, el cual nos permite obtener una probabilidad condicional a partir de la probabilidad condicional invertida y sus componentes, como podemos ver en la fórmula:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

En nuestro caso, para calcular $P(CDMX|x)$ esta fórmula se traduce a:

$$P(CDMX|x) = \frac{P(x|CDMX) \times P(CDMX)}{P(x)}$$

Para calcular fácilmente la probabilidad condicional $P(x|CDMX)$, basta con que contemos el número de veces que aparece el texto “desde el aeropuerto capitalino fue trasladada” para cada una de las etiquetas. Aquí nos topamos con un problema... ¡el texto completo no aparece en ninguna de las etiquetas!

Clasificador ingenuo

Un clasificador ingenuo es aquél que supone que las propiedades proporcionadas al algoritmo son independientes entre ellas, logrando hacerlo más robusto sobre pocos datos o datos mal etiquetados. Utilizando este supuesto, nos permite tomar cada palabra del texto como propiedad, en lugar de tomar el texto completo. Podemos ver que:

$$P(x) = P(desde) \times P(el) \times P(aeropuerto) \times P(capitalino) \times P(fue) \times P(trasladado)$$

lo cual nos permite calcular la probabilidad condicional como

$$P(x|CDMX) = P(desde|CDMX) \times P(el|CDMX) \times P(aeropuerto|CDMX)$$

$$\times P(capitalino|CDMX) \times P(fue|CDMX) \times P(trasladado|CDMX)$$

facilitándonos el cálculo de esta probabilidad, ya que estas palabras aparecen una o varias veces en nuestro conjunto de entrenamiento.

Calculando probabilidades

Como habíamos mencionado, haremos un conteo de palabras y de etiquetas. Primero, calculamos la probabilidad *a priori* de las etiquetas, suponiendo que estas son equiprobables: $P(\text{CDMX}) = \frac{1}{3}$, $P(\text{Nacional}) = \frac{1}{3}$ y $P(\text{Deportes}) = \frac{1}{3}$. Después, para calcular $P(\text{desde}|\text{CDMX})$ debemos tomar el número de veces que aparece la palabra “desde” dentro de los textos con la etiqueta “CDMX” y lo dividimos entre el número de palabras totales con la etiqueta “CDMX”, pero podemos ver que “desde” no aparece dentro de los textos con la etiqueta de “CDMX”, lo cual nos arroja que $P(\text{desde}|\text{CDMX}) = 0$, pero al hacer las multiplicaciones para determinar $P(x|\text{CDMX})$ causaría que la probabilidad del texto completo sea 0 y se pierda información sobre las probabilidades.

Para resolver este problema se aplican técnicas de suavizado (ver Manning y Schütze 1999, pág. 202) como el suavizado de Laplace, sumando 1 a cada dividendo y lo contrarrestamos sumando el número de palabras posibles sin repetirse a cada divisor, así tendríamos $P(\text{desde}|\text{CDMX}) = \frac{0+1}{35+67}$. Esto lo repetimos con cada palabra del texto “desde el aeropuerto capitalino fue trasladada” y con cada etiqueta, obteniendo las siguientes tablas:

X	$P(X \text{CDMX})$
desde	$\frac{0+1}{35+67}$
el	$\frac{2+1}{35+67}$
aeropuerto	$\frac{0+1}{35+67}$
capitalino	$\frac{2+1}{35+67}$
fue	$\frac{0+1}{35+67}$
trasladada	$\frac{0+1}{35+67}$

X	$P(X \text{Deportes})$
desde	$\frac{0+1}{31+67}$
el	$\frac{0+1}{31+67}$
aeropuerto	$\frac{0+1}{31+67}$
capitalino	$\frac{0+1}{31+67}$
fue	$\frac{0+1}{31+67}$
trasladada	$\frac{0+1}{31+67}$

X	P(X Nacional)
desde	$\frac{0+1}{30+67}$
el	$\frac{2+1}{30+67}$
aeropuerto	$\frac{0+1}{30+67}$
capitalino	$\frac{0+1}{30+67}$
fue	$\frac{0+1}{30+67}$
trasladada	$\frac{0+1}{30+67}$

Ahora, multiplicamos todas las probabilidades y obtenemos que:

$$P(x|CDMX) = \frac{1}{102} \times \frac{3}{102} \times \frac{1}{102} \times \frac{3}{102} \times \frac{1}{102} \times \frac{1}{102}$$

$$P(x|CDMX) = \frac{9}{11.26162419 \times 10^{11}} = 7.99174244 \times 10^{-12}$$

$$P(x|Deportes) = \frac{1}{98} \times \frac{1}{98} \times \frac{1}{98} \times \frac{1}{98} \times \frac{1}{98} \times \frac{1}{98}$$

$$P(x|Deportes) = \frac{1}{8.858423809 \times 10^{11}} = 1.128868997 \times 10^{-12}$$

$$P(x|Nacional) = \frac{1}{97} \times \frac{3}{97} \times \frac{1}{97} \times \frac{1}{97} \times \frac{1}{97} \times \frac{1}{97}$$

$$P(x|Nacional) = \frac{3}{8.329720049 \times 10^{11}} = 3.601561616 \times 10^{-12}$$

Con lo cual nuestro clasificador le otorga la etiqueta **CDMX** al texto “desde el aeropuerto capitalino fue trasladada” y la segunda etiqueta más probable sería **Nacional**.

Mejorando el clasificador

Una manera sencilla de mejorar la clasificación de los textos es quitando palabras vacías (*stop words*), donde una palabra vacía es aquella que “no aporta información al texto que está siendo procesado” (Manning y Schütze 1999, pág. 533). En cualquier lenguaje, estas palabras vacías sirven para proveer contexto a un texto, pero el Clasificador Bayesiano Ingenuo no hace uso del contexto para determinar las etiquetas pertenecientes a cada texto, entonces estas palabras sólo hacen más lenta la obtención de etiquetas y pueden introducir falsos positivos en la clasificación. En este caso procederíamos a quitarlas de nuestros conjuntos de entrenamiento y prueba.

Utilizando las palabras vacías que define Alir3z4 en su repositorio de GitHub¹ en el ejemplo anterior, podemos ver que los datos de entrenamiento se reducen a:

Texto	Etiqueta
acuerdo portal datos abiertos ciudad méxico	CDMX
gobierno capitalino mejora seguridad ciudad méxico	CDMX
claudia sheinbaum aseguró elevarán tarifas capitalinos	CDMX
clasificación octavos final champions league	Deportes
fifa determinó deberá pagar seis millones dólares	Deportes
oportunidad clasificar octavos, nápoles recibirá rb salzburgo	Deportes
presidente lamentó sucedido alcalde valle chalco	Nacional
comisión temporal presupuesto instituto nacional electoral	Nacional
mañana, durante conferencia prensa, presidente	Nacional

Figura 11.2 Datos de entrenamiento después de eliminar palabras vacías

Y nuestro texto a etiquetar quedaría así: "aeropuerto capitalino trasladada". Entonces, haciendo los cálculos nuevamente, podemos ver que quedan de la siguiente manera

X	P(X CDMX)
aeropuerto	$\frac{0+1}{18+51}$
capitalino	$\frac{2+1}{18+51}$
trasladada	$\frac{0+1}{18+51}$

X	P(X Deportes)
aeropuerto	$\frac{0+1}{19+51}$
capitalino	$\frac{0+1}{19+51}$
trasladada	$\frac{0+1}{19+51}$

X	P(X Nacional)
aeropuerto	$\frac{0+1}{17+51}$
capitalino	$\frac{0+1}{17+51}$
trasladada	$\frac{0+1}{17+51}$

Ahora, multiplicamos todas las probabilidades y obtenemos que:

¹<https://github.com/Alir3z4/stop-words>

$$P(x|CDMX) = \frac{1}{69} \times \frac{3}{69} \times \frac{1}{69}$$

$$P(x|CDMX) = \frac{3}{328509} = 0.000009132$$

$$P(x|Deportes) = \frac{1}{70} \times \frac{1}{70} \times \frac{1}{70}$$

$$P(x|Deportes) = \frac{1}{343000} = 0.000002915$$

$$P(x|Nacional) = \frac{1}{68} \times \frac{1}{68} \times \frac{1}{68}$$

$$P(x|Nacional) = \frac{1}{314432} = 0.00000318$$

Con esto podemos ver que es más sencillo hacer las operaciones, obtenemos probabilidades de mayor magnitud por la disminución del número de palabras y las palabras restantes nos permiten una mayor exactitud en la determinación de las etiquetas.

Desarrollo e implementación

La práctica consiste en obtener el texto de las noticias proporcionadas por el feed RSS de Aristegui Noticias, separar las noticias en dos conjuntos: de entrenamiento y de prueba, eliminar las palabras vacías haciendo uso del repositorio de GitHub proporcionado anteriormente, entrenar un Clasificador Bayesiano Ingenuo con el primer conjunto y evaluar qué tan bien predice las etiquetas de las noticias del segundo conjunto. Para lograrlo deberán usar el ejemplo anterior como guía para obtener las probabilidades de cada etiqueta presente en la base de prueba y después comparar las etiquetas obtenidas por el clasificador contra las etiquetas reales.

En el código auxiliar se encuentra el archivo `feed.db` que contiene al menos 50 notas con título, texto, etiquetas y liga con el siguiente formato:

```

1
2   Title: ...
3   Tags: ...
4   Text: ...
5   Link: ...
6

```

11. Bayes Ingenuo

Por ejemplo:

```

1   Title: Devolverá Senado 281 millones 537 mil 756 pesos a Hacienda
2   Tags: MÉXICO, Poder Legislativo, Presupuesto, Secretaría de Hacienda y ...
3   Text: El Senado de la República devolverá a la Tesorería de la Federación ...
4   Link: https://aristeguinoticias.com/1401/mexico/devolvera-senado...
5
6   Title: Consejeros y empleados del INE cobrarán lo mismo que en 2018 ...
7   Tags: MÉXICO, INE, Justicia, Poder Judicial, Presupuesto
8   Text: Una jueza federal permitió al Instituto Nacional Electoral (INE) ...
9   Link: https://aristeguinoticias.com/1401/mexico/consejeros-y-empleados...
10
11  Title: Buen inicio de semana del peso: dólar baja a 19 unidades
12  Tags: Dinero y Economía, Estados Unidos, Finanzas, Peso Dólar
13  Text: El peso mexicano tuvo este lunes un buen inicio de semana, ...
14  Link: https://aristeguinoticias.com/1401/lomasdestacado/buen-inicio...
15
16

```

Separen este archivo en dos para obtener una base de datos de entrenamiento y otra de prueba, procurando que la mayor parte de las noticias queden en la base de datos de entrenamiento.

Después deberán obtener el texto, el título y la etiqueta de cada noticia incluida en la base de datos de entrenamiento. Con esta información deberán calcular tres probabilidades, como se explica en el ejemplo visto en la sección anterior:

1. La probabilidad de que se obtenga una etiqueta en específico:

$$\frac{\# \text{ de veces que aparece la etiqueta}}{\# \text{ total de muestras de etiquetas}}$$

2. La probabilidad de que se obtenga una palabra en específico:

$$\frac{\# \text{ de veces que aparece una palabra}}{\# \text{ total de muestras de palabras}}$$

3. La probabilidad de que una palabra aparezca en el texto de alguna etiqueta:

$$P(\text{palabra} | \text{etiqueta})$$

Podemos ver que terminaremos con tres variables aleatorias, las cuales debemos conservar en Factores para agilizar su consulta. Si tienen problemas obteniendo la probabilidad de que una palabra aparezca en el texto de alguna etiqueta les recomiendo lo siguiente: crean un diccionario de palabras por cada etiqueta, que contenga el número de veces que aparece cada palabra con la etiqueta dada y el número total de palabras por etiqueta.

Una vez que tengan sus probabilidades, habrán acabado con la fase de entrenamiento de su Clasificador Bayesiano Ingenuo. Solo faltaría evaluar qué tan bien predice las etiquetas de las noticias incluidas en su base de datos de prueba. Para esto, tendrán que determinar a qué etiqueta pertenece el texto de las noticias. Deberán proporcionar las 3 más probables, y comparar con las etiquetas reales imprimiendo ambos conjuntos de etiquetas lado a lado.

En el código auxiliar también podrán encontrar un script auxiliar que obtiene las noticias y las guarda en la base de datos, que es un archivo de texto plano. Lo pueden utilizar para obtener bases de datos actualizadas.

Requisitos y resultados

Todo lo anterior lo deberán anexar a la carpeta de la práctica, listo para cargarlo y probarlo. En el `readme` deben especificar cómo se ejecuta, cómo se cargan los archivos, etc. Tampoco olviden comentar su código.

12 | Cadenas de Márkov

Verónica Esther Arriola Ríos
Benjamín Torres Saavedra

En esta práctica se implementarán funciones para realizar cálculos básicos sobre cadenas de Márkov.

Objetivo

- Que el alumno se familiarice con los procesos estocásticos discretos denominados Cadenas de Márkov discretas y realice una implementación para resolver algunas de las preguntas más frecuentes que se pueden plantear sobre estos procesos. (GEO Tutoriales 2015)

Introducción

Definición 12.1: Cadena de Márkov discreta

- Una *Cadena de Márkov* es un modelo bayesiano dinámico donde cada nodo representa el estado del sistema al tiempo t , denotado $S^{(t)}$, y éste sólo tiene un parente: el estado del sistema en el tiempo anterior S^{t-1} . [Figura 12.1]
- El sistema tiene un conjunto finito de estados discretos posibles $S = \{s_1, \dots, s_n\}$
- Asume *invarianza temporal*, es decir, la probabilidad de transición $P(S^{(t+1)}|S^{(t)})$ es la misma para todo t .

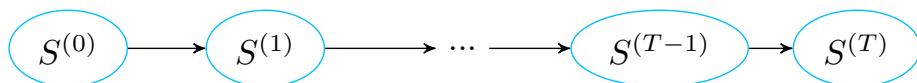


Figura 12.1 Una Cadena de Márkov es una Red de Bayes degenerada.

Inferencia

Frecuentemente la probabilidad de transición de un estado a otro se expresa con un autómata al plantear el problema. De este autómata es posible extraer el factor representando la probabilidad condicionar $P(S^{(t+1)}|S^{(t)})$, con las variables $S^{(t)}$ y $S^{(t+1)}$ en su alcance.

Si a esto agregamos el factor con la distribución de probabilidad para el estado inicial $P(S^{(0)})$, es posible obtener $P(S^{(t)})$ para cualquier t utilizando la regla de la cadena de Bayes. Por ejemplo, para $t = 1$:

$$P(S^{(1)}) = \sum_{S^{(0)}} P(S^{(1)}|S^{(0)})P(S^{(0)})$$

con lo que obtenemos la probabilidad del que el sistema se encuentre en cualquier estado al tiempo $t = 1$, independientemente de en qué estado haya iniciado.

Recursivamente:

$$P(S^{(t+1)}) = \sum_{S^{(t)}} P(S^{(t+1)}|S^{(t)})P(S^{(t)})$$

Sin embargo, utilizar factores puede ser un cañón para este caso, pues cada nodo sólo tiene un nodo padre. Podemos notar que la multiplicación de factores y marginalización sobre la variable padre, se pueden realizar en una sola operación si escribimos la distribución de probabilidad condicional en una matriz:

$$T = \begin{bmatrix} P(S = s_1|S = s_1) & \dots & P(S = s_1|S = s_n) \\ P(S = s_2|S = s_1) & \dots & P(S = s_2|S = s_n) \\ \dots \\ P(S = s_n|S = s_1) & \dots & P(S = s_n|S = s_n) \end{bmatrix}$$

y la distribución de probabilidad para el estado en cada tiempo con un vector:

$$P(S) = \begin{bmatrix} P(s_1) \\ P(s_2) \\ \dots \\ P(s_n) \end{bmatrix}$$

Entonces el cálculo de la distribución de probabilidad para otros tiempos se resume como:

$$\begin{aligned} P(S^{t+1}) &= TP(S^t) \\ P(S^{t+1}) &= T^n P(S^0) \end{aligned}$$

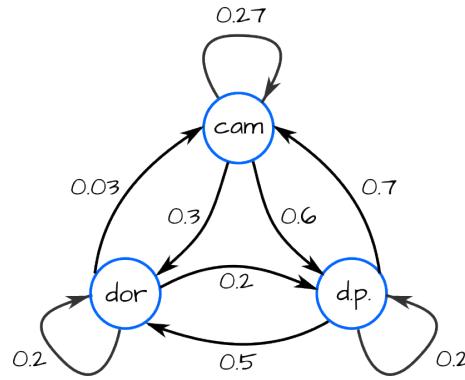


Figura 12.2 Las probabilidades de transición expresadas con un autómata.

Ejemplo: El soldado

Supongamos que estamos modelando el comportamiento de un soldado en un videojuego de acción. Este soldado está vigilando un área restringida y puede tener alguno de tres comportamientos: estar caminando, quedarse de pie o dormido. El estado del soldado queda descrito por la variable $S \in \{\text{caminando}, \text{dormido}, \text{de pie}\}$, lo cual en ocasiones abreviaremos como $S \in \{\text{cam}, \text{dor}, \text{d.p.}\}$. Para que su comportamiento no sea tan predecible, cambiará su actividad aleatoriamente según describe el autómata en la Figura 12.2.

Entonces, la matriz de transición se acomoda como en la tabla siguiente:

		$P(S^{(t+1)} S^{(t)})$		
$S^{(t+1)} \setminus S^{(t)}$		caminando	dormido	de pie
$S^{(t)}$	caminando	0.27	0.3	0.6
dormido	0.03	0.2	0.2	
de pie	0.7	0.5	0.2	

Inicialmente sea:

$S^{(0)}$	$P(S^{(0)})$
caminando	0.2
dormido	0.2
de pie	0.6

Para calcular qué es probable que esté haciendo el soldado al tiempo siguiente basta

con multiplicar:

$$\begin{aligned} P(S^{(1)}) &= TP(S^{(0)}) \\ &= \begin{bmatrix} 0.27 & 0.3 & 0.6 \\ 0.03 & 0.2 & 0.2 \\ 0.7 & 0.5 & 0.2 \end{bmatrix} \begin{bmatrix} 0.2 \\ 0.2 \\ 0.6 \end{bmatrix} \\ &= \begin{bmatrix} 0.474 \\ 0.166 \\ 0.36 \end{bmatrix} \end{aligned}$$

Estado estacionario

Definición 12.2: Cadena irreducible

Se dice que una cadena es *irreducible* si es posible acceder a cualquier estado desde cualquier otro estado, ya sea directamente o a través de un camino de nodos.

Definición 12.3: Estado periódico

“Un estado es periódico si, partiendo de ese estado, sólo es posible volver a él en un número de etapas que sea múltiplo de un cierto número entero d mayor que uno.”

“Si d = 1 decimos que el estado es *aperiódico*.” (GEO Tutoriales 2013)

En particular, un sistema donde es posible transitar desde un estado s_i a cualquier otro estado s_j directamente, da lugar a una cadena irreducible con estados aperiódicos. Cuando estas dos condiciones se cumplen, la cadena eventualmente alcanzará un *estado estacionario*, donde la distribución de probabilidad al tiempo $t + 1$ será igual a la distribución al tiempo t . Es decir, cumple con la ecuación:

$$P(S^{(t)}) = TP(S^{(t)})$$

Adicionalmente, como se trata de una distribución de probabilidad, las componentes de $P(S)$ deben sumar uno:

$$\sum_i P(s_i) = 1$$

Si el número de valores posibles para S es n , esto nos da un sistema de $n + 1$ ecuaciones linealmente independientes. Este sistema está sobre determinado, por lo que para resolverlo se realiza una aproximación utilizando mínimos cuadrados (Maplesoft 2022).

El sistema en forma matricial se escribe:

$$(T - \mathbb{1})P(S^{(t)}) = \emptyset$$

$$[11\dots1]P(S^{(t)}) = 1$$

Concatenando los renglones de ambas matrices:

$$\begin{bmatrix} T - \mathbb{1} \\ [11\dots1] \end{bmatrix} P(S^{(t)}) = \begin{bmatrix} \emptyset \\ 1 \end{bmatrix}$$

Mínimos cuadrados ajustará un plano que pase lo más cerca posible de los puntos descritos por cada renglón de la matriz que multiplica a $P(S^{(t)})$.

Ejemplo: comportamiento del soldado en el estado estacionario

Si bien el comportamiento del soldado de nuestro ejemplo mutará más al inicio, eventualmente se estabilizará en una rutina donde una sola distribución de probabilidad describirá qué podría estar haciendo. Para encontrar esta distribución el sistema que debemos resolver ajustando mínimos cuadrados es:

$$\begin{bmatrix} -0.73 & 0.3 & 0.6 \\ 0.03 & -0.8 & 0.2 \\ 0.7 & 0.5 & -0.8 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \text{cam} \\ \text{dor} \\ \text{d.p.} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

El resultado es:

$$P(S) = \begin{bmatrix} 0.42220485 \\ 0.12822518 \\ 0.44956998 \end{bmatrix}$$

y es posible verificar que $TP = P$.

La Figura 12.3 muestra los puntos descritos por esta matriz y el plano definido por la solución encontrada.

Hay más consultas que se pueden realizar con una cadena de Márkov, con los conocimientos que tienes de redes de Bayes ya te debe ser posible inferir lo que tienes que hacer. Si aún tienes dudas puedes consultar la página sobre cadenas de Márkov en https://en.wikipedia.org/wiki/Markov_chain así como algunos ejemplos sencillos en https://en.wikipedia.org/wiki/Examples_of_Markov_chains.

Desarrollo

Para esta práctica se requiere implementar una clase para trabajar con cadenas de Márkov. Se recomienda programarla en Python, ya que la biblioteca numpy contiene

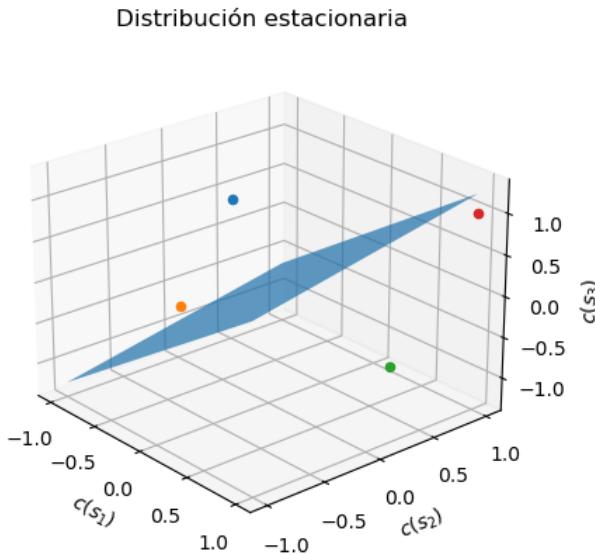


Figura 12.3 El plano ajustado corresponde a la distribución estacionaria.

métodos para trabajar con matrices y vectores, lo cual simplifica mucho los puntos siguientes. También se recomienda utilizar matrices columna para representar vectores, de esta manera funcionarán correctamente las multiplicaciones de matriz por vector.

La clase CadenaDeMarkov debe contener:

1. Un constructor que reciba como parámetros:
 - a) Una lista con los nombres los estados posibles.
 - b) Un vector con la probabilidad de iniciar en cada uno de los estados posibles.
 - c) Una matriz de probabilidades, con la probabilidad de transitar de cada estado hacia los demás.
2. Un método para generar una secuencia de estados a partir del modelo de Márkov iniciado dado el número n de elementos que tendrá la secuencia; opcionalmente puede recibir como parámetro una semilla para la generación de números aleatorios.
Para generar esta muestra necesitarás calcular los vectores con las distribuciones de probabilidad para n pasos. Dada cada distribución, utiliza un número aleatorio para determinar cuál de los estados corresponderá a ese paso.
Devuelve una lista con la secuencia de estados.
3. Obtener la probabilidad de una cadena de estados (punto extra si se permite que esta cadena tenga estados indeterminados). Observa que esta es la distribución de probabilidad conjunta $P(S_0 = s_0, S_1 = s_1, \dots, S_n = s_n)$, escrito de otra forma, $P(s_0, s_1, \dots, s_n)$. Deberá recibir como parámetro una lista con la secuencia de estados y devolver la probabilidad.

4. Estimar las probabilidades a largo plazo de cada uno de los estados, es decir, la distribución límite, cuando sea posible. Este método deberá devolver el vector con la distribución de probabilidades.
5. Agregar un archivo donde se utilice tu clase para resolver un ejemplo, usando cada uno de los métodos. Puedes usar algún ejemplo del sitio de gestión de operaciones.

Requisitos y resultados

Incluir:

1. El archivo de la clase CadenaDeMarkov.
2. El archivo con el demo de prueba.

13 | Viterbi

Víctor Germán Mijangos de la Cruz

Esta práctica introduce el concepto de Modelo Oculto de Márkov, así como sus aplicaciones en el aprendizaje supervisado de cadenas, en donde se implementará el algoritmo de Viterbi.

Objetivo

Que el alumno conozca el concepto de Modelo Oculto de Márkov, su forma de representarlos, así como sus aplicaciones. Que sea capaz de aplicar los Modelos Ocultos de Márkov a problemas de etiquetado de secuencias de forma óptima por medio del algoritmo de Viterbi.

Introducción

Los Modelos Ocultos de Márkov a los que nos referiremos como HMM (por sus siglas en inglés *Hidden Markov Models*) son un modelo gráfico *generativo*. Esto es, son capaces de aprender un modelo de la distribución de probabilidad subyacente a los datos de entrenamiento y utilizarlo para generar nuevos datos con características semejantes. Además, generalizan el modelo de Bayes Ingenuo e incorporan el concepto de procesos de Márkov en su estructura, es decir, que asumimos un sistema dinámico en el que el estado al tiempo t depende sólo del estado al tiempo $t - 1$.

En el clasificador de Bayes Ingenuo se busca estimar una probabilidad conjunta de una serie de observaciones, determinadas por un vector, x y las clases y . En el modelo de Bayes Ingenuo se busca estimar la clase \hat{y} que cumpla:

$$\hat{y} = \arg \max_y P(y, x) = \arg \max_y p(x|y)$$

Es decir, aquella clase \hat{y} que maximice la probabilidad de haber observado a x . Si x

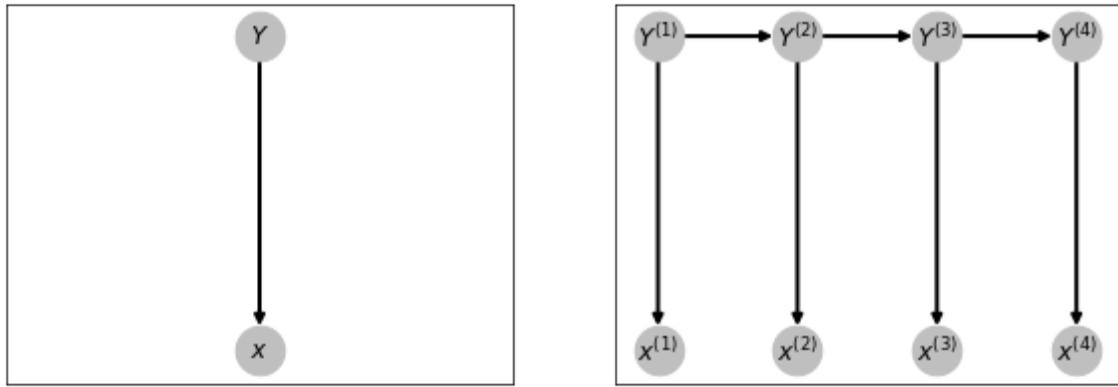


Figura 13.1 Comparación entre la gráfica del modelo de Bayes Ingenuo y la gráfica de un Modelo Oculto de Márkov.

es un vector con n elementos independientes, la probabilidad condicionada a y puede expresarse como $P(x|y) = \prod_{i=1}^n P(x_i|y)$. El modelo de Bayes Ingenuo es de gran utilidad para clasificar un vector de rasgos dentro de una clase o categoría que lo represente. Sin embargo, hay problemas, como el etiquetado de cadenas del lenguaje natural, donde lo que se busca es pasar de una cadena a otra. Por ejemplo, si en una cadena del lenguaje natural como: "yo salto el salto", queremos identificar los tipos de palabras (nombre común, verbo, preposición, pronombre, etc.), no basta con clasificar cada palabra según su clase más probable pues, como se puede observar en este ejemplo, la etiqueta depende del contexto. Así, la palabra "salto" es en el primer caso un verbo (ya que es precedido por un pronombre) y en el segundo un nombre común (antecedido por el artículo).

Los HMMs son una solución a este tipo de problemas. Podemos pensar a un HMM como un modelo de Bayes Ingenuo dinámico, donde las clases o categorías conforman un proceso de Márkov. En la Figura 13.1 se puede observar la comparación entre una gráfica del modelo de Bayes Ingenuo (a la izquierda) y el modelo oculto de Márkov (a la derecha). El HMM genera transiciones entre las cadenas de salida, para así poder estimar la pertinencia de una categoría en referencia no sólo a una observación, sino a las categorías previas de la cadena. En términos generales, definimos una HMM como sigue:

Definición 13.1: Modelo Oculto de Márkov

Un *Modelo Oculto de Márkov* o HMM es un modelo bayesiano definido como una 5-tupla $HMM = (X, Y, A, \Pi, B)$ cuyos elementos son:

- X es un conjunto de *observables* que dependen de Y .
- Y es un conjunto de *emisiones* que definen un proceso de Márkov.

- A es la matriz de transiciones y Π es un vector de probabilidades iniciales; ambos definen el proceso de Márkov sobre Y .
- B es una matriz de probabilidades de observaciones, que contiene las probabilidades de las observaciones condicionadas a las emisiones.

Para obtener la cadena de emisiones $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ más óptima, que etiquete a la cadena de entrada $\hat{x}_1, \hat{x}_2, \dots, \hat{x}_n$, se debe optimizar la probabilidad conjunta:

$$\begin{aligned}\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n &= \arg \max_{y_1, \dots, y_n} P(Y^{(1)} = y_1, Y^{(2)} = y_2, \dots, Y^{(n)} = y_n, x^{(1)}, x^{(2)}, x^{(n)}) \\ &= \arg \max_{y_1, \dots, y_n} \prod_{t=1}^n P(Y^{(t)} = y_t | Y^{(t-1)} = y_{t-1}) P(x^{(t)} | Y^{(t)} = y_t) \quad (13.1)\end{aligned}$$

Con la convención de que $P(Y^{(1)} = y_1) := P(Y^{(1)} = y_1 | Y^{(0)})$ es la probabilidad inicial del proceso de Márkov en Y . Esto establece los HMM y da forma al problema que buscamos resolver.

Estimación del modelo

Un HMM puede considerarse un modelo de aprendizaje; es decir, un modelo en que se deben aprender ciertos parámetros a partir de un conjunto de datos. Con estos datos, se deben obtener los valores que pueden tomar Y y X , así como las probabilidades de transición (A), iniciales (Π) y de observaciones (B). Al proceso de obtener los parámetros del modelo se le conocerá como el *entrenamiento del modelo*.

Tomemos un caso en donde queremos obtener cadenas en un alfabeto $Y = \{a, b, c\}$ a partir de un alfabeto binario $X = \{0, 1\}$. Tenemos entonces que estimar un modelo para el proceso de Márkov en Y determinado por las transiciones, la probabilidad de que se pase de un símbolo en Y a otro, y las probabilidades iniciales. Pero además, tenemos que estimar las probabilidades de observaciones, esto es, la probabilidad de observar un símbolo binario asociado a un símbolo del alfabeto $\{a, b, c\}$.

Podemos asumir que las probabilidades de transición en A son las siguientes:

		$P(Y^{(t)} Y^{(t-1)})$		
$Y^{(t)} \setminus Y^{(t-1)}$		a	b	c
$Y^{(t-1)}$	a	0.1	0.7	0.6
	b	0.5	0	0.2
	c	0.4	0.3	0.2

Y consideraremos los siguientes iniciales Π :

y_0	$P(Y^{(0)})$
a	0.4
b	0.3
b	0.3

Como se puede observar, el HMM define un proceso de Márkov sobre Y, pero además integra un modelo de predicción; consideremos entonces las probabilidades de observaciones en B dadas como:

		$P(x^{(t)} Y^{(t)})$		
		a	b	c
$x^{(t)} \setminus Y^{(t)}$		0	1	2
0		0.6	0.3	0.2
1		0.4	0.7	0.8

Esta matriz de observaciones guarda similitudes con el modelo de Bayes Ingenuo, pues se puede decir que determina las probabilidades condicionadas a su clase. En la forma gráfica del modelo, expuesta en la Figura 13.1 (derecha), las probabilidades de transición en A determinan las aristas horizontales que van de una variable $Y^{(t)}$ hacia la siguiente, mientras que la matriz B determina las aristas verticales entre los elementos $Y^{(t)}$ y $x^{(t)}$.

Si bien el modelo gráfico que define a las HMMs está determinado por la gráfica secuencia de la Figura 13.1, es común representar a los parámetros del modelo en una gráfica, que se define como sigue:

1. Se tiene un nodo raíz que indica el comienzo.
2. Desde el nodo raíz se generan aristas hacia los símbolos en Y con las probabilidades iniciales.
3. Se generan aristas entre todos los símbolos en Y que representan las probabilidades de transición.
4. De los símbolos en Y se generan aristas hacia los símbolos en X con las probabilidades de observación.

Por ejemplo, del modelo anterior se obtiene la gráfica de la Figura 13.2, la cual resume de manera visual el modelo $HMM = (X, Y, A, \Pi, B)$. Sin embargo, cuando se tienen muchos símbolos este tipo de gráficas se hacen complejas y no es conveniente realizarlas.

Etiquetado con HMMs

Para realizar el etiquetado de una cadena de entrada $x^{(1)}, x^{(2)}, \dots, x^{(n)}$ dado un HMM, se requiere encontrar la cadena que solucione el problema de optimización de la Ecuación 13.1. Sin embargo, como se está computando un máximo, se deben explorar todos

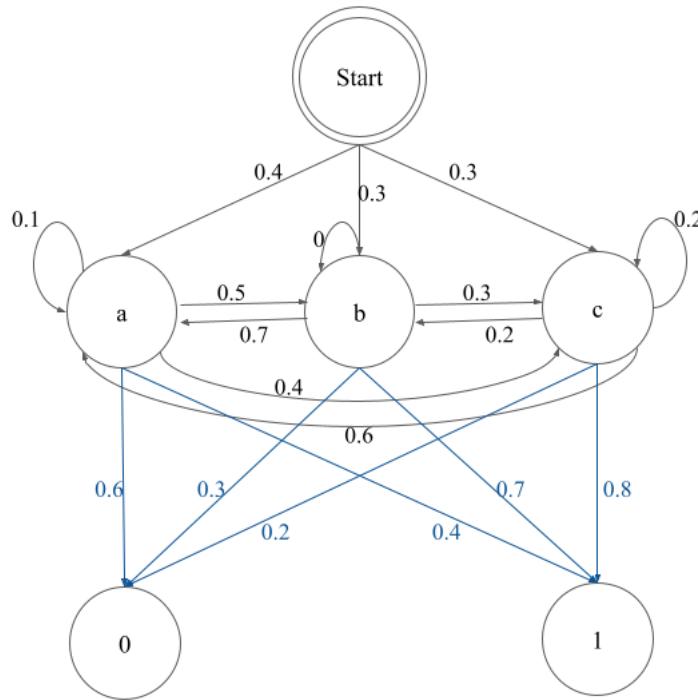


Figura 13.2 Gráfica que representa los parámetros del HMM.

los símbolos posibles y_t para cada uno de los estados de la cadena. Así, por ejemplo, si $|Y| = M$ es el número de símbolos de emisión, se tendrá que calcular M probabilidades para una cadena de longitud 1. Para una cadena de longitud 2, el primer símbolo de entrada podrá tomar M valores, y el segundo también podrá tomar estos M valores, por lo que se tendrán que calcular M^2 combinaciones, para de ahí poder tomar la que haya dado la probabilidad más alta. En general, encontrar el argumento que maximiza la probabilidad sobre las cadenas de emisión tiene una complejidad de $O(M^n)$.

En problemas reales, como etiquetado del lenguaje natural, donde se cuenta con cadenas de varios cientos de símbolos, y cadenas que pueden tener la extensión de un documento, el cálculo de estas probabilidades se vuelve intratable. En la Figura 13.3 se muestra un diagrama, conocido como diagrama de Trellis, que muestra los posibles formas de etiquetar la cadena “yo salto el salto de altura” con cinco símbolos de emisión.¹ Partiendo de cualquier símbolo de inicio se puede tomar cualquier camino, lo que genera un total de $5^6 = 15625$ posibles caminos, o formas de etiquetar la cadena.

Algoritmo de Viterbi

Para solventar la complejidad del cálculo de las probabilidades en un HMM, se utilizará un algoritmo dinámico, el conocido como algoritmo de Viterbi. Este algoritmo busca que

¹Los símbolos corresponden a las etiquetas DA =Determinante Artículo, NC =Nombre Común, PP =Preposición, DP =Determinante Pronombre y V =Verbo.

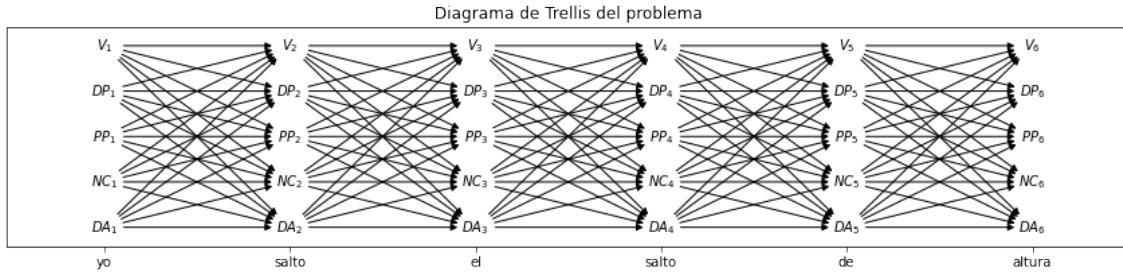


Figura 13.3 Diagrama de Trellis que muestra todos los caminos posibles para etiquetar una cadena de longitud 6 con 5 emisiones.

en cada tiempo de la cadena se obtengan sólo los caminos más probables que vayan hacia los símbolos en el tiempo actual. Es decir, busca maximizar el camino que lleva a un símbolo en el tiempo t .

Entonces el algoritmo de Viterbi genera M caminos finales (donde M es el número de símbolos de emisión) en lugar de M^n . De estos caminos toma el más probable como aquel que contiene los símbolos de emisión que maximizan la probabilidad en el modelo.

De forma intuitiva, en cada tiempo t de la cadena de entrada, se obtiene la probabilidad que maximiza el camino hacia cada símbolo y_i en ese tiempo. Para esto, se guardan las variables:

$$\delta_i(t+1) = \max_{y_{i_1} \dots y_{i_t}} P(Y^{(1)} = y_{i_1}, \dots, Y^{(t)} = y_{i_t}, x^{(1)}, \dots, x^{(t+1)}, Y^{(t+1)} = y_i)$$

Estas variables consideran el camino $y_{i_1} \dots y_{i_t}$ con mayor probabilidad para llegar al símbolo y_i en el tiempo $t + 1$. Estas variables sólo guardan la probabilidad, los símbolos de emisión se almacenan en otra variable ϕ . Finalmente, cuando se ha terminado de recorrer la cadena, se toma el camino que maximiza la probabilidad total de la cadena como la salida para etiquetar la cadena de entrada. En el Algoritmo 4 se condensa el algoritmo de Viterbi.

En este caso, se utilizan el vector de iniciales (π_i es la probabilidad inicial del símbolo i), las probabilidades de transición A ($A_{i,k}$ es la probabilidad de pasar del símbolo k al símbolo i) y de emisiones B ($B_{x^{(t)}, i}$ es la probabilidad de que la observación $x^{(t)}$ esté generada por el símbolo i). El algoritmo de Viterbi reduce el número de caminos a considerar; por ejemplo, la Figura 13.3 puede reducir el número de caminos considerablemente a sólo 5, como se observa en la Figura . De hecho, se puede observar que en cada iteración el algoritmo sólo compara el número de símbolos de emisión M con los mismos símbolos en el estado anterior. Esto implica que cada iteración realiza M^2 cálculos, por lo que la complejidad del algoritmo para una cadena de longitud n es de $O(nM^2)$.

Algoritmo 4 Algoritmo de Viterbi

```

function VITERBI(cadena, HMM)
     $x^{(1)}, x^{(2)}, \dots, x^{(n)} \leftarrow \text{SPLIT}(\text{cadena})$ 
     $\delta_i(1) \leftarrow p(x^{(1)}|y_i)p(y_i) = B_{x^{(1)}, i} \pi_i$   $\triangleright \forall y_i \in Y$ 
    for  $t \leftarrow 2$  a  $n$  do
         $\delta_i(t+1) \leftarrow \max_k p(x^{(t+1)}|y_i)p(y_i|y_k)\delta_k(t) = \max_k B_{x^{(t+1)}, i} A_{i,k} \delta_k(t)$ 
         $\phi_i(t+1) \leftarrow \arg \max_k B_{x^{(t+1)}, i} A_{i,k} \delta_k(t)$ 
    end for
     $\hat{y}_n \leftarrow \arg \max_i \delta_i(n)$ 
    for  $t \leftarrow n-1$  a  $1$  do
         $\hat{y}_t \leftarrow \phi_{\hat{y}_{t+1}}(t+1)$ 
    end for
    return  $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ 
end function

```

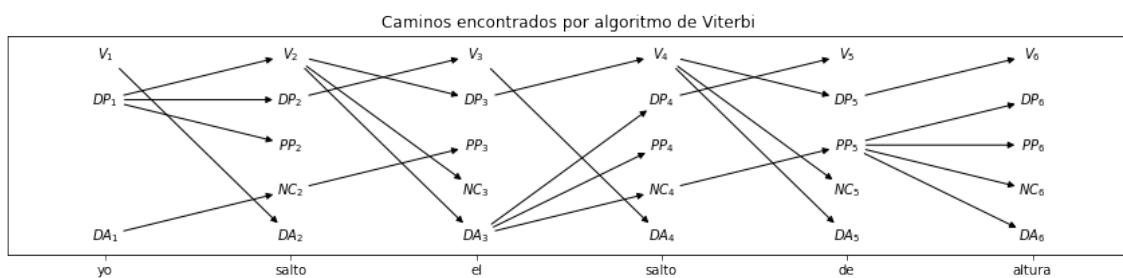


Figura 13.4 Gráfica de los caminos después de aplicar el algoritmo de Viterbi

Etiqueta	Descripción	Ejemplo
DA	Determinante Artículo	el, la, los, las
DP	Determinante Pronombre	yo, tú, ella
PP	Preposición	de, ante, para
NC	Nombre Común	gato, perro, flor
V	Verbo	comer, llover, dar

Tabla 13.1 Conjunto de etiquetas para etiquetado gramatical

Ejemplo: etiquetado del lenguaje natural

Consideremos que queremos etiquetar las categorías gramaticales en lenguaje natural. Esto es, dada una cadena de lenguaje natural, queremos saber cuál es la cadena de etiquetas que le corresponde según las palabras que componen la entrada. Para hacer esto tomaremos algunos símbolos convencionales para etiquetas gramaticales. Estos se muestran en la Tabla 13.1.

También podemos tomar palabras del español que correspondan a estas etiquetas, aquí nos limitaremos a algunas de ellas; en particular, usaremos las palabras: 'el', 'la', 'yo', 'ellos', 'de', 'salto', 'altura', 'cuerda', 'vino', 'tomaban' y 'saltaban'. Podemos conformar nuestro modelo. Asumiremos que ya hemos estimado las probabilidades.

El vector de iniciales será:

y_0	$P(Y^{(0)})$
DA	0.18
DP	0.5
PP	0.08
NC	0.16
V	0.08

Consideremos la siguiente matriz de probabilidades de transición:

		$P(Y^{(t)} Y^{(t-1)})$					
		DA	DP	PP	NC	V	
$Y^{(t)}$	$Y^{(t-1)}$	DA	0.1	0.1	0.3	0.05	0.4
		DP	0.1	0.1	0.1	0.05	0.1
PP		0.1	0.1	0.1	0.8	0.1	
NC		0.6	0.1	0.4	0.05	0.3	
V		0.1	0.6	0.1	0.05	0.1	

Finalmente, la matriz de probabilidades de emisión estará dada como:

$Y^{(t)} \setminus Y^{(t-1)}$	$P(Y^{(t)} Y^{(t-1)})$				
	DA	DP	PP	NC	V
el	0.23	0.05	0.05	0.01	0.05
la	0.23	0.05	0.05	0.01	0.05
yo	0.06	0.27	0.05	0.01	0.05
ellos	0.06	0.28	0.05	0.05	0.05
de	0.06	0.05	0.55	0.05	0.05
salto	0.06	0.05	0.05	0.3	0.25
cuerda	0.06	0.05	0.05	0.25	0.05
vino	0.06	0.05	0.05	0.3	0.05
tomaban	0.06	0.05	0.05	0.01	0.2
saltaban	0.06	0.05	0.05	0.01	0.2

Esto define nuestro HMM, por lo que ahora podemos aplicarlo a resolver un problema de etiquetado. Tomemos una cadena simple; sea esta: 'ellos tomaban vino'. Procedemos entonces a aplicar el algoritmo de Viterbi. Consideramos que los símbolos de entrada son $x^{(1)} = \text{ellos}$, $x^{(2)} = \text{tomaban}$ y $x^{(3)} = \text{vino}$. Procedemos entonces a realizar el algoritmo:

1. Calculamos los valores $\delta_i(1)$ para los símbolos iniciales, aquí se toma en cuenta el observable $x^{(1)} = \text{ellos}$; estos son:

$$\begin{aligned}\delta_{\text{DA}}(1) &= 0.06 \cdot 0.18 = 0.01 \\ \delta_{\text{DP}}(1) &= 0.28 \cdot 0.5 = 0.14 \\ \delta_{\text{PP}}(1) &= 0.05 \cdot 0.08 = 0.004 \\ \delta_{\text{NC}}(1) &= 0.01 \cdot 0.16 = 0.001 \\ \delta_{\text{V}}(1) &= 0.05 \cdot 0.08 = 0.004\end{aligned}$$

2. Ahora generamos los caminos para el siguiente símbolo en la cadena correspondientes a la observación $x^{(2)} = \text{tomaban}$:

δ	DA	DP	PP	NC	V	máx	ϕ
$\delta_{\text{DA}}(2)$	6.0e-05	8.4e-04	7.2e-05	3.0e-06	9.6e-05	8.4e-04	DP
$\delta_{\text{DP}}(2)$	5.0e-05	7.0e-04	2.0e-05	2.5e-06	2.0e-05	7.0e-04	DP
$\delta_{\text{PP}}(2)$	5.e-05	7.e-04	2.e-05	4.e-05	2.e-05	7.e-04	DP
$\delta_{\text{NC}}(2)$	6.0e-05	1.4e-04	1.6e-05	5.0e-07	1.2e-05	1.4e-04	DP
$\delta_{\text{V}}(2)$	2.00e-04	1.68e-02	8.00e-05	1.00e-05	8.00e-05	1.68e-02	DP

Observemos que, en este caso, para todas las posibles emisiones en $t = 2$, la emisión en el tiempo anterior que produjo la mayor probabilidad fue DP. En general, no necesita ser el caso.

3. Obtenemos los casos para el último estado correspondientes a la observación $x^{(3)} = \text{vino}$:

δ	DA	DP	PP	NC	V	máx	ϕ
$\delta_{DA}(3)$	5.040e-06	4.200e-06	1.260e-05	4.200e-07	4.032e-04	4.032e-04	V
$\delta_{DP}(3)$	4.2e-06	3.5e-06	3.5e-06	3.5e-07	8.4e-05	8.4e-05	V
$\delta_{PP}(3)$	4.2e-06	3.5e-06	3.5e-06	5.6e-06	8.4e-05	8.4e-05	V
$\delta_{NC}(3)$	1.512e-04	2.100e-05	8.400e-05	2.100e-06	1.512e-03	1.512e-03	V
$\delta_V(3)$	4.2e-06	2.1e-05	3.5e-06	3.5e-07	8.4e-05	8.4e-05	V

En este caso todos los caminos son de la forma DP - V. Esto genera un camino, que corresponde a la parte de avance del algoritmo. Ahora tenemos que obtener el retroceso, el cuál nos dará las etiquetas. La última etiqueta corresponderá al camino que tenga mayor probabilidad, que en este caso corresponde a $\delta_{NC}(3)$. Por tanto, la última etiqueta será NC. De aquí recuperaremos los valores de ϕ que corresponden al camino, que en este caso es V y DP. Por tanto, las etiquetas finales serán:

$$DP - V - NC$$

Es decir, nuestra cadena 'ellos tomaban vino' tiene las etiquetas de pronombre, verbo y nombre común.

Desarrollo

En esta práctica se implementará el algoritmo de Viterbi para etiquetado de categorías gramaticales de una cadena de lenguaje natural. Tómese el HMM del ejemplo anterior, en donde los símbolos de emisiones están dados por:

$$Y = \{DA, DP, PP, NC, V\}$$

Y los símbolos de observación se determinan por:

$$X = \{el, la, yo, ellos, de, salto, cuerda, vino, tomaban, saltaban\}$$

Utilizamos las probabilidades que se han considerado previamente. El código fuente se puede revisar en el archivo `HMMViterbi.ipynb`.

Implementación

Para la implementación del código del algoritmo de Viterbi para el etiquetado de cadenas de lenguaje natural se propone generar la función de avance y de retroceso, para incorporarlas dentro de una función `ViterbiParser`. Se sugiere que se utilicen estructuras de datos de arreglos de `numpy`, para trabajar con productos de matrices y vectores, que pueden realizar los cálculos de manera más eficiente. La implementación constará de los siguientes elementos:

- Se tomará como entrada los parámetros del HMM; las probabilidades pueden representarse por medio de arreglos. Para poder manejarlos adecuadamente, se requiere indexar los símbolos de emisión y observación. Se puede usar una estructura de hash o diccionario para relacionar los símbolos y sus índices. Esta indexación es arbitraria, y sólo debe tomarse en cuenta que debe haber una correspondencia entre los índices y las entradas de los arreglos.
- El método tomará como entrada una cadena del lenguaje natural. Esta cadena debe ser preprocesada. Ya que se están considerando sólo palabras en minúsculas, cualquier cadena debe contener sólo minúsculas y deberán eliminarse símbolos ortográficos en caso de presentarse. Posteriormente, se generarán los tokens, que corresponden a las palabras, separando por espacios en blanco para conformar una lista. Finalmente, se recomienda sustituir los tokens en las cadenas por sus índices numéricos correspondientes.
- La inicialización del algoritmo de Viterbi almacena la variable δ en el tiempo 1, que corresponde al producto de las probabilidades de la observación condicionadas por los iniciales. Esto puede expresarse como el producto punto a punto (que denotamos \odot) entre el vector renglón de B correspondiente a la primera observación por el vector de iniciales. Matemáticamente:

$$\delta(1) \leftarrow B_{x^{(1)}, \cdot} \odot \Pi$$

Donde $\delta(1)$ es un vector con entradas $\delta_i(1)$ correspondientes a los símbolos. Este producto se expresa en python por el operador `x*y`, donde `x` e `y` son arreglos.

- Para el paso de avance se obtendrán los valores que maximicen los caminos hacia la emisión en el estado actual. Esto se obtiene para todas las emisiones. Ya que se debe computar los valores $B_{x^{t+1}, i} \delta_k(t)$ se puede tomar un producto externo, para generar una matriz que contenga estos productos y posteriormente se multiplique por la matriz de transición A . Finalmente, se obtendrán los valores máximos sobre los renglones de la matriz resultante. Se recomienda usar una variable p que contenga sólo las probabilidades; esta variable se computará entonces como:

$$p \leftarrow A \odot \left(B_{x^{(t+1)}, \cdot} \otimes \delta(t) \right)$$

Aquí \otimes representa el producto externo, el cual se expresa en python por medio de la paquetería numpy como `numpy.outer()`. De esta forma se pueden obtener tanto los valores para la variable δ (valores máximos) como para la variable ϕ (las etiquetas o argumentos que maximizan las probabilidades). La variable ϕ guardará las emisiones que terminarán etiquetando a la cadena. Los máximos y los mínimos pueden obtenerse con los métodos `numpy.max()` y `numpy.argmax()`.

- Una vez concluido el paso de avance, el paso de retroceso comenzará generando la última etiqueta; es decir, la etiqueta que corresponde a la última palabra de la cadena de entrada y posteriormente avanzará hacia atrás sobre las variables ϕ para

generar la cadena de etiquetas de emisión. Para concluir se revertirá la cadena para obtener las etiquetas en el orden correcto. Si se ha trabajado con índices numéricos, se recuperarán las etiquetas correspondientes a este índice.

- El etiquetador en base a HMMs y Viterbi se probará sobre diferentes cadenas del lenguaje que contengan las palabras en las observaciones. Se sugiere probar en la cadena 'yo salto el salto de altura', donde se puede comprobar la calidad del etiquetado.

Requisitos y resultados

El sistema de etiquetado deberá ser una función que pueda trabajar con cualquier cadena que contenga los tokens (palabras) consideradas en las observaciones. En caso de observar un token que no está se podrá ignorar o emitir un mensaje de error. Se espera que la salida sea una cadena con las etiquetas correspondientes a las categorías gramaticales de los tokens en la entrada. El código deberá estar adecuadamente comentado. Se tomarán en cuenta los siguientes resultados.

1. Que el manejo de las cadenas de lenguaje natural y etiquetas de emisión sea adecuado.
2. Que la representación de los datos se maneje adecuadamente; en particular, que las operaciones para el algoritmo de Viterbi entre las matrices y los vectores sean eficientes.
3. Que tanto la entrada como la salida sean cadenas interpretables por el usuario humano.

PARTE II

Proyectos

14 | STRIPS

Verónica Esther Arriola Ríos

En este proyecto implementarás una pequeña máquina de inferencias utilizando los conocimientos de la primera parte del curso.

Meta

El alumno experimentará de primera mano cómo programar los elementos esenciales de una máquina de inferencias para un sistema experto.

Objetivos

- Que el alumno se familiarice con el uso del lenguaje de descripción PDDL para expresar los pormenores de un problema de planeación.
- Familiarizarse con el uso del diseño orientado a objetos en el código, para facilitar la implementación de los algoritmos que resolverán el problema de planeación.
- Que el alumno implemente un algoritmo de búsqueda que trabaje sobre objetos en un dominio PDDL.
- Que el alumno valore la relevancia del algoritmo de unificación, en un caso sencillo, para la identificación de acciones aplicables a un estado del dominio.

Desarrollo

En este paquete se te entrega:

- Una copia de la gramática BNF del lenguaje para descripción de problemas de planeación PDDL. No utilizarás todas las opciones que se incluyen, así que sólo revísala para que te des una idea de lo que se puede hacer.

- Un script de Python con la definición de las clases que representarán a los objetos en el dominio, hechos, acciones posibles, estado inicial y meta de un problema de planeación. Lee con cuidado la documentación. Estas clases corresponden a los elementos de PDDL que vas a utilizar. Observa que las clases listan los atributos, pero esta representación es independiente de la notación con paréntesis de PDDL.

Escenario de prueba

El programa que vas a realizar deberá poder resolver un pequeño problema de planeación, definido según la notación de PDDL. El dominio siguiente se puede utilizar para probarlo.

```

1  ;; Especificación en PDDL1 de un dominio DWR simplificado
2
3  (define (domain platform-worker-robot)
4      (:requirements :strips :typing)
5      (:types
6          contenedor
7          pila           ;tiene una base y una pila de contenedores.
8          grúa          ;sostiene máximo un contenedor.
9      )
10     (:predicates
11         (sostiene ?k - grúa ?c - contenedor) ;la grúa ?k sostiene al
12         ↪ contenedor ?c
13         (libre ?k - grúa)                      ;la grúa ?k está libre
14         (en ?c - contenedor ?p - pila)        ;el contenedor ?c está en
15         ↪ la pila ?p
16         (hasta_arriba ?c - contenedor ?p - pila) ;el contenedor ?c se
17         ↪ encuentra hasta arriba de la pila ?p
18         (sobre ?k1 - contenedor ?k2 - contenedor);el contenedor ?k1 está
19         ↪ sobre el contenedor ?k2
20     )
21     ;; toma un contenedor de una pila con la grúa
22     (:action toma
23         :parameters (?k - grúa ?c - contenedor ?p - pila)
24         :vars (?otro - contenedor)           ;variables locales (extra
25         ↪ a PDDL)
26         :precondition (and (libre ?k) (en ?c ?p)
27                           (hasta_arriba ?c ?p) (sobre ?c ?otro))
28         :effect (and (sostiene ?k ?c) (hasta_arriba ?otro ?p)
29                     (not (en ?c ?p)) (not (hasta_arriba ?c ?p))
30                     (not (sobre ?c ?otro)) (not (libre ?k))))
31
32     ;; pone al contenedor sostenido por la grúa en una pila
33     (:action pon
34         :parameters (?k - grúa ?c - contenedor ?p - pila)
35         :vars (?otro - contenedor)           ;variables locales
36         :precondition (and (sostiene ?k ?c) (hasta_arriba ?otro ?p))
37         :effect (and (en ?c ?p) (hasta_arriba ?c ?p) (sobre ?c ?otro))

```

14. STRIPS

```

33           (not (hasta_arriba ?otro ?p)) (not (sostiene ?k ?c)
34             ↵)
35           (libre ?k))))
```

A continuación se incluye un ejemplar de problema. Puedes diseñar otros tú y verificar que tu programa también los resuelva correctamente.

```

1 ; ; un problema sencillo del dominio DWR simplificado
2 (define (problem dwrbp1)
3   (:domain platform-worker-robot)
4
5   (:objects
6     k1 k2 - grúa
7     p1 q1 p2 q2 - pila
8     ca cb cc cd ce cf pallet - contenedor)
9
10  (:init
11    (en ca p1)
12    (en cb p1)
13    (en cc p1)
14
15    (en cd q1)
16    (en ce q1)
17    (en cf q1)
18
19    (sobre ca pallet)
20    (sobre cb ca)
21    (sobre cc cb)
22
23    (sobre cd pallet)
24    (sobre ce cd)
25    (sobre cf ce)
26
27    (hasta_arriba cc p1)
28    (hasta_arriba cf q1)
29    (hasta_arriba pallet p2)
30    (hasta_arriba pallet q2)
31
32    (libre k1)
33    (libre k2))
34
35    ; ; el problema consiste en mover todos contenedores
36    ; ; ca y cc a la pila p2, los demás a q2
37  (:goal
38    (and (en ca p2)
39      (en cb q2)
40      (en cc p2)
41      (en cd q2)
42      (en ce q2)
43      (en cf q2))
44  )
```

Tareas a realizar

Las tareas que debes realizar son las siguientes:

Parte I

1. Dibuja el estado inicial. Puedes incluirlo como comentario en tu código como arte ASCII o adjuntar un archivo de imagen (no debe ser un archivo pesado).
2. Crea a mano los objetos correspondientes al dominio y problema anteriores. Imprímelos y observa que todo esté correcto.
3. Observa que el estado del mundo se representa como una lista de predictados aterrizados. Agrega el código necesario (clases o funciones) para que, dado un estado determine si una acción es aplicable o no y con qué sustitución (qué objeto se asigna a qué variable).
4. Agrega el código para determinar si un estado satisface las condiciones indicadas en el campo *meta*.
5. Incluye código con pruebas que muestren que tus implementaciones son correctas.

Parte II

6. Ahora agrega el código necesario para realizar una búsqueda en amplitud de la secuencia de acciones que se debe seguir para alcanzar la meta. Haz que imprima lo que está haciendo de modo que puedas ver si se comporta como deseas.
7. Utiliza tu programa para buscar una solución al problema anterior. ¿Es suficiente información? ¿Cómo se comporta tu programa?

15 | Lego Mindstorms

Luis Alfredo Lizárraga Santos

Objetivo

Que el estudiante trabaje con un robot programable de tal forma que comprenda las funciones de distintos tipos de sensores y motores.

Introducción

Se trabajará con Lego Mindstorms ya que es una plataforma fácil de usar y no es necesario tener conocimientos de circuitos digitales ni analógicos para la construcción del robot.

¿Qué es Lego Mindstorms?

Lego Mindstorms es un paquete de software y hardware que permite crear robots programables, modulares y personalizables. Son caracterizados por contener una computadora central y módulos como: motores, actuadores, sensores de luz, de color, de presión, infrarrojos, etc. Y, como son Lego, se puede tener muchas configuraciones gracias a sus piezas intercambiables y construibles.

Ahora, para la práctica necesitaremos algunas cosas extras, ya que se utilizará el lenguaje de programación Python, en lugar de usar la aplicación de diseño propia de Mindstorms. Se supondrá que se trabaja en Linux.

Instalando ev3dev

ev3dev es una distribución de Linux basada en Debian, lo cual permite tener una mayor disponibilidad de lenguajes y bibliotecas. La única desventaja es que sólo es compatible

con la versión EV3 de Lego Mindstorms.

Se necesitará:

1. El bloque programable de Mindstorms.
2. Una memoria microSD con capacidad menor o igual a 32GB, preferentemente vacía ya que se borrarán todos los datos de esta.
3. Una computadora con un adaptador para tarjetas microSD.
4. Y una manera de comunicarse con el bloque programable, ya sea:
 - Por un cable USB
 - Por Wi-Fi usando un adaptador USB
 - Por Ethernet usando un adaptador USB
 - Por Bluetooth

Pueden obtener la imagen de ev3dev en: <https://github.com/ev3dev/ev3dev/releases>. Deben descargar la imagen que comienza con “ev3”, ya que las demás son para otros sistemas.

Una vez que tengan la imagen, procederemos a descomprimirla y copiarla a la memoria microSD:

1. Abran una terminal y ejecuten el comando `df` y guarden el resultado.
2. Inserten la memoria y vuelven a ejecutar `df` para obtener el nombre asignado por el sistema a su memoria.
3. Desmonten la memoria, ejecutando `sudo umount /dev/sdb1` (suponiendo que su memoria se encuentra en `/dev/sdb1`).
4. Ahora, copiarán la imagen directamente a su memoria usando las aplicaciones `dd` y `xzcat`. Ejemplo: `xzcat /Downloads/ev3dev-yyyy-mm-dd.img.xz | sudo dd bs=4M of=/dev/sdb`.
5. Y por último, en cuanto termine la ejecución del paso anterior (puede demorar algunos minutos, no se desesperen), faltaría ejecutar `sync` para asegurarse de que todas las escrituras al disco terminen y podrán remover la memoria microSD de su computadora.

Después de haber descomprimido y escrito la imagen a la memoria, basta con insertarla al bloque programable y encenderlo.

Conectándose a internet

Para lograr una conexión a internet (y poder acceder al bloque por medio de ssh), pueden consultar <http://www.ev3dev.org/docs/getting-started/> donde encontrarán guías para cada método: por USB, usando un adaptador inalámbrico, usando un adaptador Ethernet USB o por bluetooth.

Instalando Python, virtualenv y bibliotecas extras

¿Qué es virtualenv?

Virtualenv es una aplicación que facilita hacer proyectos con Python, ya que crea un ambiente virtual en el que pueden utilizar una versión de Python específica, junto con *add-ons* para esa versión, sin tener que utilizar esa misma versión globalmente. Léase: crea un contenedor con Python independiente.

Instalación

Una vez establecida una conexión por ssh al bloque programable, basta ejecutar lo siguiente:

1. apt-get update
2. apt-get install virtualenv virtualenvwrapper python-setuptools python-smbus python-pil
3. source /etc/bash_completion.d/virtualenvwrapper
4. mkvirtualenv {nombre del contenedor} -python=/usr/bin/python2.7 -system-site-packages
5. workon {nombre del contenedor}
6. easy_install -U python-ev3

Con esto hecho, ya podrán acceder a los motores y sensores del EV3 desde Python, importando la biblioteca ev3.

Para poder ejecutar su aplicación, pueden desarrollarla directamente desde el bloque programable usando vi o nano, pero se recomienda generar el archivo aparte y después copiarlo al bloque utilizando el comando scp. Por ejemplo:

Suponemos que tenemos un archivo llamado `seguidor_de_linea.py`, entonces ejecutamos:

- scp seguidor_de_linea root@[dir_ip_bloque]:/home/robot

Con esto, usando ssh, nos dirigimos a la carpeta raíz del usuario y ejecutamos:

- python seguidor_de_linea.py

Desarrollo e implementación

Su práctica consiste en crear un robot seguidor de línea y un algoritmo para éste.

Robot

Pueden utilizar el sensor de color (para identificar la línea) y los distintos motores que tienen a su disposición. El diseño del robot es libre, un ejemplo de cómo montar los motores y sensor lo pueden ver en la figura 15.1.

Algoritmo

Aquí tienen un ejemplo que utiliza dos motores (también lo pueden encontrar en el apéndice A.1), ubicados en los costados del bloque programable, y el sensor de color para ubicarse. Cabe notar que este algoritmo es bastante sencillo, ustedes deberán mejorararlo para que su robot siga la línea lo más fluidamente posible.

```

1 from ev3.ev3dev import Motor
2 from ev3.lego import ColorSensor
3
4 a = Motor(port=Motor.PORT.A) # Abre el motor en el puerto A
5 b = Motor(port=Motor.PORT.D) # Abre el motor en el puerto D
6 sense = ColorSensor() # Abre el sensor para leer los valores
7 # que obtiene
8
9 def move():
10     maxSpeed = 100
11     black = 10
12     white = 78
13     speedA = 50
14     speedB = 50
15     midpoint = (white-black)/2 + black

```

15. Lego Mindstorms

```
16     tolerance = 20
17     last_error = 0
18
19     while True:
20         # Obtiene el valor de reflectividad de la superficie
21         # que ve el sensor
22         val = sense.reflect
23         error = midpoint - val
24         print error, val
25
26         if abs(error) < tolerance:
27             speedA = maxSpeed - last_error/2
28             speedB = maxSpeed - last_error/2
29             last_error = error
30         else:
31             if error > 0:
32                 speedB = speedB + error/2
33                 speedA = speedA - error/2
34                 last_error = error
35             else:
36                 error = abs(error)
37                 speedB = speedB - error/2
38                 speedA = speedA + error/2
39                 last_error = error
40
41         if speedA < 0: speedA = 0
42         if speedA > maxSpeed: speedA = maxSpeed
43         if speedB < 0: speedB = 0
44         if speedB > maxSpeed: speedB = maxSpeed
45
46         # Hace que el motor A gire a la velocidad especificada
47         # indefinidamente
48         a.run_forever(speedA)
49         # Hace que el motor B gire a la velocidad especificada
50         # indefinidamente
51         b.run_forever(speedB)
```

Lamentablemente no hay mucha documentación sobre este plug-in, pero siempre pueden consultar más ejemplos en: <https://github.com/topikachu/python-ev3>

Requisitos y resultados

Deben entregar un script o aplicación que resuelva el problema de seguir una línea negra u obscura en un fondo contrastante junto con imágenes de la construcción del robot. Su script deberá ser lo más eficiente posible y debe guiar al robot a una velocidad mayor que el ejemplo que aparece en la sección anterior.

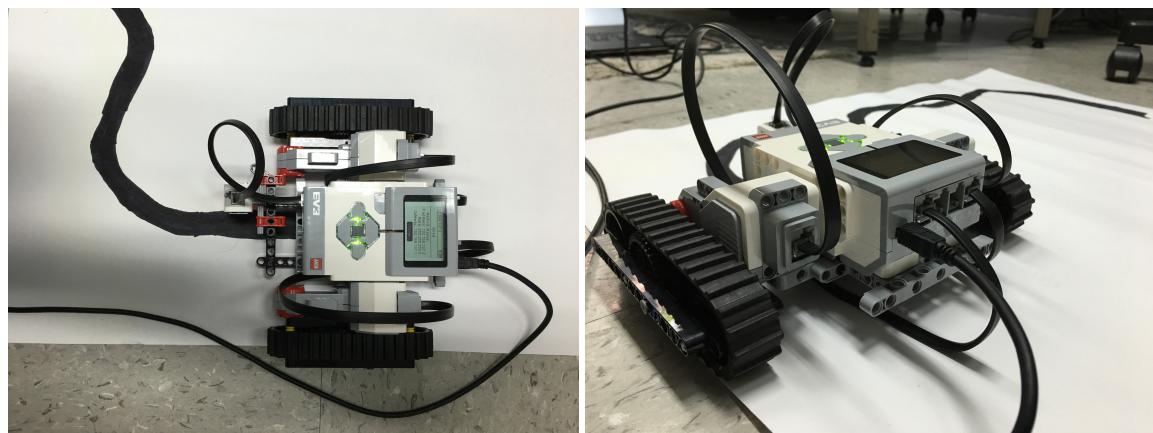


Figura 15.1 Aquí se muestra una configuración ejemplo de robot seguidor de línea utilizando Lego Mindstorms.

16 | Localización de robots

Luis Alfredo Lizárraga Santos
Verónica Esther Arriola Ríos

Objetivo

Conocer e implementar una técnica de localización utilizada en aplicaciones reales, específicamente para los robots *Rhino* y *Minerva* en los museos *Deutsches Museum Bonn* y *National Museum of American History*, respectivamente (Fox, Burgard y Thrun 1999). Se utilizará Processing como herramienta de visualización de la técnica de localización.

Introducción

“La localización robótica ha sido considerado como uno de los problemas fundamentales de la robótica. El objetivo de la localización es lograr estimar la posición del robot en un ambiente, apoyándose en un mapa de éste y con lecturas de sensores” (Fox, Burgard y Thrun 1999). Tomando como base esta publicación de Dieter Fox, et. al. conocaremos los tipos de localización que existen y presentaremos la técnica que desarrolla la publicación: la localización de Markov.

Localización Robótica

Las técnicas que se han desarrollado hasta la fecha tratan de resolver uno de los dos tipos de localización que hay:

- *Localización local o rastreo*. Estas técnicas tratan de compensar errores odométricos durante el movimiento del robot. Son técnicas auxiliares que refinan la estimación que se tiene de la posición del robot en el entorno todo el tiempo, si la pierden es casi imposible volver a recuperarla (Fox, Burgard y Thrun 1999).

- *Localización global.* Estas técnicas están diseñadas para encontrar la posición estimada del robot globalmente, es decir, no es necesario tener un aproximado inicial de su posición. Estas técnicas pueden resolver el problema de localizar al robot al momento de encenderlo, al igual que permiten que se lleve al robot a una posición aleatoria del entorno durante su operación y recuperar su posición (Fox, Burgard y Thrun 1999).

Como se podrán dar cuenta, las técnicas de localización global son más poderosas que las locales. Para esta práctica desarrollaremos una técnica de localización global de Markov.

Localización de Markov

La localización de Markov utiliza un sistema probabilístico que mantiene una distribución de probabilidad de la posición del robot sobre todo el entorno. Es decir, la probabilidad de que el robot se encuentre en cada posición del entorno en un tiempo dado. Por ejemplo, el robot puede iniciar con una distribución de probabilidad uniforme representando que no tiene idea de dónde se encuentra en el entorno, esto es, cada posición en el entorno tiene la misma probabilidad de que el robot se encuentre en ella. En el caso en el que el robot esté muy seguro de su posición, la distribución de probabilidad se convierte en una distribución unimodal centrada en la posición del robot.

Ejemplo

Estudiaremos el ejemplo sencillo que presentan Dieter Fox et. al. para ilustrar los conceptos de la localización de Markov.

Consideremos el entorno mostrado en la figura 16.1. Para simplificar, asumamos que el robot sólo puede moverse en una dimensión (enfrente-atrás). Ahora, supongamos que posicionamos el robot en algún lugar aleatorio del entorno, pero no le informamos al robot cuál es su posición. La localización de Markov representa este estado de “confusión” como una distribución de probabilidad uniforme sobre todo el conjunto de posibles posiciones del entorno, como lo muestra la primer gráfica de la figura 16.1. Asumamos que el robot hace una medición con sus sensores y determina que está al lado de una puerta. La localización de Markov modifica la distribución de probabilidad de tal manera que las posiciones que se encuentran a un lado de puertas tengan mayor probabilidad, esto queda ilustrado en la segunda gráfica de la figura 16.1. Notemos que la distribución resultante es multimodal⁽¹⁾, ya que la información obtenida por los sensores es insuficiente para determinar exactamente la posición del robot. También notemos que las posiciones que no se encuentran a un lado de una puerta aún tienen una probabilidad mayor que cero, esto es porque las mediciones de los sensores contienen ruido. Ahora, supongamos que

¹Multimodal: Que tiene varios puntos máximos.

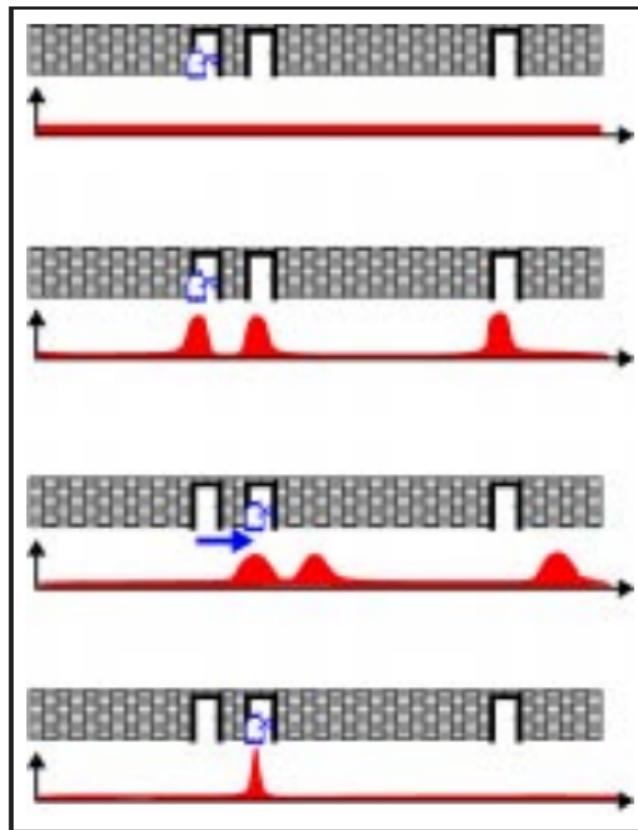


Figura 16.1 Ejemplo de localización de Markov. La gráfica de color rojo representa la distribución de probabilidad. (Fox, Burgard y Thrun 1999)

el robot se mueve un metro hacia el frente. La localización de Markov incorpora este movimiento para que la distribución de probabilidad cambie como se representa en la tercer gráfica de la figura 16.1. Finalmente, asumamos que el robot vuelve a tomar una medición con sus sensores. Incorporando esta información a la obtenida anteriormente, vemos cómo la distribución de probabilidad cambia (cuarta gráfica de la figura 16.1) para asignar una alta probabilidad a la posición del robot, mientras que todas las demás posiciones tienen una probabilidad casi nula (Fox, Burgard y Thrun [1999](#)).

Desarrollo

Tomando como base el modelo de Markov que presentan Dieter Fox et. al., el desarrollo de la práctica se dividirá en dos partes: la simulación del ambiente en donde se encuentra el robot y del robot, y la implementación del modelo en este ambiente.

Simulación del ambiente y del robot

Primero, iniciemos con el contenido y especificación del mundo en el cual se moverá el robot. Este mundo contendrá obstáculos y será rectangular (imaginense un cuarto o una oficina), las paredes serán obstáculos y cualquier otra cosa que se pueda encontrar en un cuarto u oficina normal: sillas, mesas, escritorios, etc. Será representado por una matriz de celdas, donde cada celda será cuadrada, podrá ser obstáculo o no y tendrá la longitud de un lado especificado en cm. En la figura 16.2 pueden apreciar un ejemplo de cómo se vería el mundo del robot representado gráficamente, donde los cuadros color café son obstáculos, el robot es el círculo verde y se muestra la orientación de este con una línea negra.

El robot podrá ocupar cualquier posición del plano de Processing², no estará necesariamente en el centro de alguna celda. De hecho, se necesitará una función que, dada la posición del robot, nos permita saber dentro de qué celda se encuentra. Esta función nos será útil para la implementación del modelo de Markov discretizado.

Movimiento del robot

El robot podrá moverse libremente en el ambiente, salvo que se encuentre de frente un obstáculo, o los bordes del mundo ya que son considerados paredes. Este movimiento será controlado por el usuario, ya sea que utilicen las flechas direccionales de su teclado, alguna otra combinación de teclas, botones en la interfaz, o el mecanismo que ustedes prefieran. Lo importante es que el robot pueda moverse en línea recta y girar.

²Cabe resaltar que para Processing el eje y se encuentra invertido, esto es, si aumentan el valor de y en el código, en la visualización esto se reflejará como si estuvieran caminando hacia abajo.

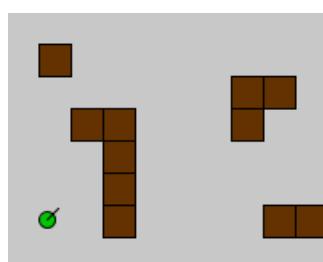


Figura 16.2 Representación visual del ambiente.

El usuario deberá poder especificar qué tanto se moverá el robot al momento de pulsar la tecla o botón requerido, ya sea si se desea cambiar de dirección o moverse en línea recta. Esto será muy importante, ya que necesitan saber la distancia real recorrida y la distancia real de giro para simular las medidas de los sensores.

Simulación de sensores

Nuestro robot tendrá tres sensores:

1. láser: mide la distancia desde el robot hacia el primer obstáculo en línea recta del sensor. El robot tendrá 8 de estos, ubicados cada 45° .
2. odométrico: mide la distancia recorrida por el robot (en cm).
3. de giro: mide el giro del robot en grados.

Como se darán cuenta más adelante, utilizamos distribuciones Gaussianas para modelar las lecturas de los sensores. Cuando alguien busca armar un robot que tenga sensores, lo primero que debe hacer es calibrarlos para obtener un umbral de error inherente al sensor que se está utilizando para tenerlo en cuenta al momento de programar rutinas o desarrollar el software que permita al robot llegar a su objetivo. Si uno grafica las medidas obtenidas por el sensor versus la medida real, se llega a apreciar que forma una campana centrada en el valor de la medida real ya que los errores pequeños son comunes, pero errores grandes no tanto. Así nosotros podemos suponer que los sensores que estamos modelando generarán mediciones parecidas y utilizamos una distribución de probabilidad normal centrada en el valor real de lo que se mide para obtener la probabilidad de que el robot se encuentre en cierta posición dadas las mediciones recibidas.

Láser En el caso de este sensor, las mediciones obtenidas pueden cambiar debido a los materiales donde se refleja el láser, entonces utilizaremos ruido gaussiano para simular este error.

Para cada sensor se necesitarán dos parámetros: la media (μ_{laser}) y la desviación estándar (σ_{laser}). La desviación estándar es un parámetro que tú debes fijar, éste representa la cantidad de ruido que se introducirá a la medición. Y la media estará dada por la distancia real medida desde el robot hacia el primer obstáculo en línea recta en la dirección del sensor.

Para simular la medición del sensor con ruido, se hará lo siguiente:

1. Obtener un número aleatorio utilizando `nextGaussian()` de la clase `java.util.Random`.
2. Multiplicar el número aleatorio obtenido por la desviación estándar (σ_{laser}).
3. Sumar a la media (μ_{laser}) el resultado anterior.

Odométrico La manera de simular la medición con ruido de éste sensor es casi idéntica al sensor láser, sólo que en lugar de medir la distancia hacia el obstáculo más cercano usaremos la distancia que el usuario proporcionó para mover al robot. Por ejemplo: Si el usuario decidió mover al robot 10cm, la media ($\mu_{\text{odométrico}}$) sería 10cm y la desviación estándar ($\sigma_{\text{odométrico}}$) un parámetro especificado por ti.

De giro Similarmente al sensor odométrico, la media ($\mu_{\text{de giro}}$) sería los grados reales girados y la desviación estándar ($\sigma_{\text{de giro}}$) un parámetro especificado por ti.

Implementación del modelo de Markov

Representación interna del mundo

El modelo de Markov en el que se basa esta práctica utiliza un modelo del mundo discretizado (como el que usamos para definir área libres y obstáculos). *Dentro de la mente del robot* el mundo será representado por una matriz de tres dimensiones, ya que tenemos tres grados de libertad para el movimiento del robot: $< x, y, \theta >$, donde θ es el ángulo hacia donde está viendo el robot. Estas tres direcciones estarán discretizadas.

Recapitulando, en la mente del robot el mundo será representado por un arreglo tridimensional $< x, y, \theta >$ donde en las coordenadas $< x, y >$ se tendrá la representación del mundo con celdas, como se especificó al inicio de la sección pasada. Si el robot se encuentra dentro de una celda, x y y serán las coordenadas del centro de esta. Y en cuanto al eje θ , θ tomará valores desde 0° hasta 315° , en aumentos de 45° . Para cada uno de esos ángulos se tendrá una representación del plano $< x, y >$ pero con el robot mirando en la dirección discretizada θ . En la figura 16.3 se aprecia un ejemplo de cómo se

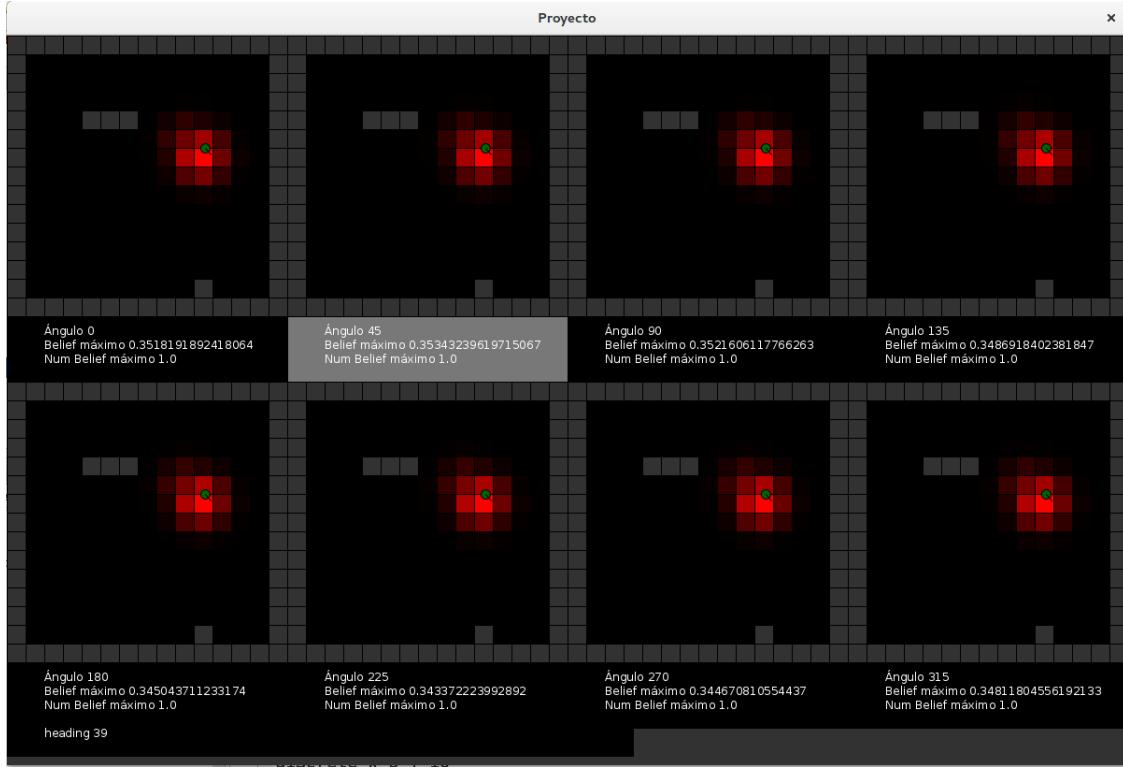


Figura 16.3 Representación del mundo en la mente del robot.

vería gráficamente la representación del mundo para el robot, con una copia del mundo por cada coordenada $< x, y, \theta >$. Como puedes ver, la visión del mundo que tiene el robot es mucho más simple de lo que realmente es, y esto puede conducir a que cometa errores.

Como habíamos mencionado antes, se necesitará una función que permita obtener la celda en donde se encuentra el robot dadas las coordenadas de éste, pero también se requerirá otra función que nos diga en qué dirección discretizada se encuentra viendo el robot. La función anterior deberá calcular el ángulo al que se encuentra viendo el robot, pero centrado sobre las posibles direcciones de movimiento. Por ejemplo: si el robot se encuentra con una dirección de 12° , la función deberá regresar el valor 0° , ya que 12° se encuentra entre $(24.5^\circ, -24.5^\circ]$.

Inicialización

Deberán posicionar al robot aleatoriamente en el mundo, después deberán calcular la probabilidad de que el robot se encuentre en alguna celda, llamémosle a esta probabilidad *creencia* y representa lo que el robot *cree* acerca de su posición en el mundo (Fox, Burgard y Thrun 1999). Al inicio esta probabilidad será igual para toda posición porque el robot no sabe dónde está:

$$P(\text{posición}) = \frac{1}{\# \text{ de posiciones que no son obstáculo}}$$

Después, deberán calcular la distancia desde el centro de cada celda al obstáculo más cercano en línea recta, para cada ángulo discretizado. Esta información la deberán guardar para cada posición de tal manera que eviten estar haciendo cálculos cada que sea considerada en el algoritmo siguiente.³

Notación

Definamos la posición discretizada del robot usando una variable tridimensional $l = <x, y, \theta>$, donde x y y son las coordenadas del centro de la celda donde se encuentra el robot y θ es la dirección discretizada en la que está viendo el robot. A partir de este momento consideraremos únicamente variables discretizadas. Sea l_t la posición real del robot en el tiempo t , y L_t la variable aleatoria que modela la probabilidad de que el robot se encuentre en una posición l dada.

Como el robot no sabe su posición exacta, su *creencia* o $Bel(L_t)$ es una distribución de probabilidad $P(L)$ sobre el espacio de posibles posiciones. Esta *creencia* nos permite saber cuál es la probabilidad de que se encuentre en una celda l en el tiempo t , formalmente $Bel(L_t = l)$. Sea n el número de posiciones posibles.

También aprovecharé para definir a las lecturas de los sensores como: s_T la lectura del láser, θ_T la lectura del giro y a_T la lectura del sensor odometrónico. Sean:

$P(s_T | l)$ es la probabilidad de que se haya obtenido una medición s_T si el robot se encuentra en la posición l . Calcularemos esta probabilidad como:

$$P(s_T | l) = \frac{1}{(\sqrt{2\pi}\sigma_{\text{láser}2})} * e^{\left(\frac{-(s_T - \mu_{\text{láser}2})^2}{2\sigma_{\text{láser}2}^2}\right)} \quad (16.1)$$

donde $\sigma_{\text{láser}2}$ modela, en parte, el error del sensor y el error por la discretización del mundo, y $\mu_{\text{láser}2}$ es la distancia desde el centroide de la celda de la posición l hacia el primer obstáculo.

$P(\theta | \theta', \theta_T)$ es la probabilidad de que el robot esté mirando en el ángulo θ de la posición l dado que se encontraba mirando en el ángulo θ' de la posición l' y se giró un ángulo θ_T . Supondremos que al girar, el robot no cambia de celda. Calcularemos esta probabilidad como:

$$P(\theta | \theta', \theta_T) = \frac{1}{(\sqrt{2\pi}\sigma_\theta)} * e^{\left(\frac{|\theta_T - (\theta - \theta')|^2}{(\sigma_\theta)^2}\right)} \quad (16.2)$$

³Se sugiere guardar esta información como un atributo de la celda

16. Localización de robots

donde σ_θ modela, en parte, el error del sensor y el error por la discretización del mundo. Esta variable será definida por ustedes.

$P(l | l', a_T)$ es la probabilidad de que el robot esté en la posición l dado que se encontraba en la posición l' y se avanzó a_T cms. Calcularemos esta probabilidad como:

$$P(l | l', a_T) = \frac{1}{(\sqrt{2\pi}\sigma)} * e^{-\frac{[(x' + a_T \cos \theta' - x)^2 + (y' + a_T \sin \theta' - y)^2 + (\theta' - \theta)^2]}{\sigma^2}} \quad (16.3)$$

con σ modelando el error del sensor.⁴

Una vez definida la notación, procederemos a ver el algoritmo completo de esta técnica de localización.

Algoritmo

El algoritmo es el siguiente:

```

for all posicion l en mundo do                                ▷ Inicialización
    Bel(L0 = l) ← P(L0 = l) =  $\frac{1}{n}$ 
end for
for all posicion l en mundo do
    determinar las distancias hacia obstáculos
end for
while true do                                              ▷ Se recibió comando o medición del láser
    if no se movió el robot then                               ▷ Se recibió medición del láser
         $\alpha_T \leftarrow 0$                                          ▷ Constante de normalización
        for all posicion l en mundo do
            Bel(LT = l) ← P(sT | l) * Bel(LT-1 = l)
             $\alpha_T \leftarrow \alpha_T + Bel(L_T = l)$ 
        end for
        for all posicion l en mundo do                          ▷ Ahora se normaliza la creencia
            Bel(LT = l) ←  $\alpha_T^{-1} * Bel(L_T = l)$ 
        end for
    end if
    if se movió el robot then
        if se obtuvo  $\theta_T$  then                                ▷ Se obtuvo una lectura del sensor de giro
            for all posicion l en mundo do
                Bel(LT = l) ← [ $\sum_{\theta'} P(\theta | \theta', \theta_T) * Bel(L_{T-1} = l)$ ]
            end for
        end if
    
```

⁴Recuerden utilizar radianes, ya que las funciones de Java están definidas en radianes.

```

if se obtuvo  $a_T$  then           ▷ Se obtuvo una lectura del sensor odométrico
    for all posición  $l$  en mundo do
         $Bel(L_T = l) \leftarrow [\sum_{l'} P(l | l', a_T) * Bel(L_{T-1} = l')]$ 
    end for
    end if
end if
end while

```

(Fox, Burgard y Thrun 1999)

Desarrollo e implementación

Tips

- Tengan bien identificado cuál es su eje X, Y y θ en las matrices utilizadas para la simulación.
- Utilicen coordenadas polares para todo cálculo, y sólo utilicen grados para visualización, esto les permitirá tener un mayor grado de exactitud y evitan gastar en convertir grados a coordenadas polares en cada operación.
- Presten mucha atención en las fórmulas utilizadas, ya que algunas piden sumar la probabilidad sobre un eje, otras piden sumar sobre todas las posiciones del mundo.
- Normalicen la creencia cuando el robot gire o se mueva, no solo cuando se reciba una medición del láser. Esto les permitirá compensar errores de redondeo al hacer los cálculos correspondientes.
- Hagan una copia del mundo antes de realizar cualquier cálculo, ya que requieren tener la creencia para cada posición en el tiempo $t-1$ para calcular t . O guarden las creencias recién calculadas en una matriz auxiliar, para luego copiarlas al mundo.

Requisitos y resultados

Esto se implementará en processing, recuerden que debe estar bien hecho y comentado, ya que esta práctica vale el 20 % de su calificación. Su aplicación deberá mostrar el mundo, los obstáculos y la probabilidad de cada celda para cada ángulo usando distintos colores o gradientes de un mismo color.

Todo lo anterior lo deberán anexar a la carpeta de la práctica, listo para cargarlo y probarlo. En el `readme` deben especificar cómo se ejecuta, cómo se cargan los archivos, etc. Tampoco olviden comentar su código.

PARTE III

Apéndices

A | Código de prácticas

Lego Mindstorms

```
1 from ev3.ev3dev import Motor
2 from ev3.lego import ColorSensor
3
4 a = Motor(port=Motor.PORT.A) # Abre el motor en el puerto A
5 b = Motor(port=Motor.PORT.D) # Abre el motor en el puerto D
6 sense = ColorSensor() # Abre el sensor para leer los valores
7 # que obtiene
8
9 def move():
10     maxSpeed = 100
11     black = 10
12     white = 78
13     speedA = 50
14     speedB = 50
15     midpoint = (white-black)/2 + black
16     tolerance = 20
17     last_error = 0
18
19     while True:
20         # Obtiene el valor de reflectividad de la superficie
21         # que ve el sensor
22         val = sense.reflect
23         error = midpoint - val
24         print error, val
25
26         if abs(error) < tolerance:
27             speedA = maxSpeed - last_error/2
28             speedB = maxSpeed - last_error/2
29             last_error = error
30         else:
31             if error > 0:
32                 speedB = speedB + error/2
33                 speedA = speedA - error/2
34                 last_error = error
35             else:
36                 error = abs(error)
37                 speedB = speedB - error/2
38                 speedA = speedA + error/2
39                 last_error = error
40
41         if speedA < 0: speedA = 0
42         if speedA > maxSpeed: speedA = maxSpeed
43         if speedB < 0: speedB = 0
44         if speedB > maxSpeed: speedB = maxSpeed
45
```

A. Código de prácticas

```
46      # Hace que el motor A gire a la velocidad especificada
47      # indefinidamente
48      a.run_forever(speedA)
49      # Hace que el motor B gire a la velocidad especificada
50      # indefinidamente
51      b.run_forever(speedB)
```

Bibliografía

- Drachman, David A. (2005). «Do we have brain to spare?» En: *Neurology* 64. URL: <https://doi.org/10.1212/01.WNL.0000166914.38327.BB>.
- Fox, Dieter, Wolfram Burgard y Sebastian Thrun (nov. de 1999). «Markov Localization for Mobile Robots in Dynamic Environments». En: *Journal of Artificial Intelligence* 11, págs. 391-427.
- GEO Tutoriales (17 de sep. de 2013). *Clasificación de Estados de una Cadena de Markov en Tiempo Discreto*. URL: <https://www.gestiondeoperaciones.net/cadenas-de-markov/clasificacion-de-estados-de-una-cadena-de-markov-en-tiempo-discreto/>.
- (31 de ago. de 2015). *Cadenas de Markov (Ejercicios Resueltos)*. URL: <https://www.gestiondeoperaciones.net/cadenas-de-markov/cadenas-de-markov-ejercicios-resueltos/>.
- Herculano-Houzel, Suzana (2009). «The human brain in numbers: a linearly scaled-up primate brain». En: *Frontier in Human Neuroscience* 9, págs. 3-31. URL: <https://doi.org/10.3389/neuro.09.031.2009>.
- Koller, Daphne y Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA.
- Lester, Patrick (2003). *A* Pathfinding para Principiantes*. URL: <http://www.policyalmanac.org/games/articulo1.htm>.
- Manning, Christopher e Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.
- Maplesoft (9 de jun. de 2022). *Computing the Steady-State Vector of a Markov Chain*. URL: <https://www.maplesoft.com/support/help/maple/view.aspx?path=examples/SteadyStateMarkovChain#:~:text=To%20compute%20the%20steady%20state,has%20components%20summing%20to%201..>
- Negnevitsky, Michael (2005). *Artificial Intelligence: A guide to Intelligent Systems*. Pearson Education Limited.
- Resnick, M. (1994). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, Cambridge, MA.
- Russell, Stuart y Peter Norving (2010). *Artificial Intelligence, A Modern Approach*. Ed. por Michael Hirsch. 2a. Pearson Prentice Hall.
- Stecanella, Bruno (mayo de 2017). *A practical explanation of a Naive Bayes classifier*. <https://monkeylearn.com/blog/practical-explanation-naive-bayes->

BIBLIOGRAFÍA

classifier/. URL: <https://monkeylearn.com/blog/practical-explanation-naive-bayes-classifier/>.