INTRODUCCION A LA INTELIGENCIA ARTIFICIAL

En la siguiente imagen se pueden observar ocho posibles definiciones para lo que llamamos inteligencia artificial:

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Históricamente, los cuatro campos en que se divide han sido seguidos por diversos autores. El test de Turing, propuesto por Alan Turing, fue diseñado para proveer una definición racional de la inteligencia. Se dice que una computadora pasa el test si una persona, luego de realizar un cuestionario, no puede deducir si quien respondió ha sido otra pe rsona o una computadora. Para ello, esta última debe contar con cuatro características:

- Procesamiento del lenguaje natural: para comunicarse fluidamente.
- Representación del conocimiento: para almacenar lo que sabe o escucha.
- Razonamiento automático: para usar la información almacenada a fin de responder preguntas y obtener nuevas conclusiones.
- Aprendizaje de máquina (o machine learning): para adaptarse a las nuevas circunstancias y detectar patrones.

Para pasar el test en su totalidad, además, la computadora necesitará visión para percibir objetos, y robótica, para manipular los mismos.

Breve historia de la IA: los campos que permitieron su origen

Es posible afirmar que la IA está fundada sobre las bases de la filosofía, las matemáticas, la economía, la neu rociencia, la psicología, la ingeniería en computación, la robótica y la lingüística.

Los filósofos consideraron que la mente es en cierta manera como una máquina, que opera sobre el conocimiento mediante alguna forma de lenguaje, a fin de poder elegir qué acciones tomar. La matemática brindó a la IA las herramientas para manipular sentencias lógicas y probabilísticas, sentando las bases de la comprensión computacional y el razonamiento acerca de algoritmos. Los economistas formalizaron el problema de la toma de decisiones que maximicen el resultado esperado. La neurociencia descubrió numerosos hechos sobre el trabajo cerebral, y la forma en que este es similar y a la vez diferente a una computadora. Los psicólogos adoptaron la idea de que humanos y animales pueden ser considerados como maquinas de procesamiento de información; la lingüística demostró que el lenguaje es lo que da vida a este modelo. La ingeniería en computación hizo posible la implementación de aplicaciones de IA creando maquinas cada vez más poderosas, similar a la robótica.

La IA ha avanzado más velozmente en las décadas pasadas, debido a los intensos usos del método científico para experimentar y comparar las aproximaciones. Sin embargo, ha tomado mayor integración en los últimos tiempos, con su implementación en sistemas reales y su vinculación con numerosas otras ciencias y campos humanos.

AGENTES INTELIGENTES

Agentes y ambientes

Un agente es cualquier cosa que puede percibir su entorno mediante sensores y actuar en consecuencia de lo que percibe mediante sus actuadores. Las percepciones son entradas que el agente capta o recibe en un instante dado. Una secuencia de percepciones es el conjunto de todas las percepciones que el agente consiguió.

Cualquier decisión que el agente tome en un momento determinado depende de esa secuencia, y no se ve afectado por nada en el entorno que no haya sido percibido por el agente.

Para ello, existe una función matemática, llamada función de agente, que mapea una secuencia de percepciones con una acción. El programa de agente es una implementación de la función de agente que se ejecuta en una arquitectura dada.

Es importante distinguir ambas cosas, la función de agente es una descripción matemática abstracta, el programa de agente es una implementación concreta, corriendo en un sistema.

Racionalidad

Un agente racional es aquel que hace las cosas bien. Hablando conceptualmente, cualquier entrada considerada por la función de agente es considerada buena. Pero ¿qué significa hacer lo correcto? Cuando un agente es colocado en un ambiente, genera una serie de acciones de acuerdo a sus percepciones. Esta secuencia de acciones provoca que el ambiente se "mueva" a través de una serie de estados. Si esta última secuencia es deseable, el agente se desempeña correctamente. Esto es lo que se conoce como medida de rendimiento o performance.

La medida de performance, entonces, indica el grado de éxito de un agente, y se mide en base a los estados del ambiente, y cabe destacar, no del agente.

Podemos afirmar que la racionalidad depende de:

- La medida de performance, que define los criterios de éxito.
- El conocimiento previo que el agente tiene del ambiente o mundo.
- Las acciones que el agente puede realizar.
- La secuencia de percepciones.

Definición de agente racional

Para cada posible secuencia de percepciones, un agente racional debe elegir una acción que se espera maximice la medida de rendimiento, dada la evidencia provista por la secuencia de percepciones y cualquier conocimiento previo que el agente posea.

Omnisciencia, aprendizaje y autonomía

Debemos ser cautelosos al momento de distinguir entre la omnisciencia y la racionalidad. Un agente omnisciente sabe cuál va a ser el verdadero resultado de sus acciones, y actúa en consecuencia. La omnisciencia es un concepto imposible en la realidad. La racionalidad maximiza el rendimiento esperado, mientras que la perfección maximiza el rendimiento real.

Nuestra definición de racionalidad, entonces, no necesita de la omnisciencia. Realizar acciones a fin de modificar las percepciones futuras (proceso generalmente llamado recolección de información) es una parte importante de la racionalidad. Sin embargo, afirmamos que un agente racional no solo recolecta información sino que aprende todo lo que puede de aquello que percibe. El aprendizaje se define como la capacidad de generar nuevo conocimiento a partir de las percepciones, y siempre se espera que un agente racional sea capaz de aprender.

Por último, la capacidad que tiene un agente de no depender solamente de conocimientos previos se conoce como autonomía. Un agente racional debe ser autónomo, aprendiendo para poder compensar sus conocimientos anteriores parciales o incorrectos. Cuando un agente consigue la suficiente experiencia de su entorno, puede volverse efectivamente independiente de su conocimiento pasado.

Naturaleza de los ambientes

En párrafos anteriores hablamos del rendimiento, el ambiente, los actuadores y los sensores. Agrupamos todos estos conceptos en lo que se conoce como PEAS (Performance, Environment, Actuators, Sensors). Esta noción sirve para definir el ambiente donde el agente va situarse.

Los ambientes pueden clasificarse de la siguiente manera:

- Completamente observable vs parcialmente observable: si los sensores del agente le dan acceso al estado total del ambiente en cualquier momento, decimos que este último es totalmente observable, en otras palabras, si los sensores captan todos los aspectos relevantes para la toma de acciones. Son convenientes porque el agente no necesita guardar ningún estado internamente para moverse por el mundo. Un ambiente puede ser parcialmente observable si los sensores están por ejemplo, dañados. Si el agente no tiene sensores, se dice que el ambiente es no observable, algo que de todas formas no determina que el objetivo sea inalcanzable.
- Un único agente vs multiagente: Cabe destacar que el segundo puede a su vez dividirse en ambientes competitivos (un ajedrez) o cooperativos (conductor de taxi).
- Determinístico vs estocástico: si el siguiente estado que adoptará el ambiente luego de una acción ejecutada por un agente, depende totalmente del estado actual y de dicha acción, decimos que el ambiente es determinístico. En cualquier otro caso, será estocástico.
  Decimos que un ambiente es incierto si no es totalmente observable o es no determinístico. Un entorno no determinístico es aquel en el que las acciones se caracterizan porque tiene varias salidas posibles, pero no llega a ser estocástico mientras no interviene en él cierto grado de probabilidad.
- Episódico vs secuencial: en el primero la experiencia del agente se divide en episodios atómicos, cada uno de los cuales brindan a al agente percepciones para que realice acciones simples, pero el siguiente episodio no depende de las acciones tomadas en los anteriores, a diferencia de los secuenciales.
- Estático vs dinámico: si el estado del ambiente puede cambiar mientras el agente "delibera", se trata de un entorno dinámico, si esto no ocurre será estático. Si el ambiente no cambia por sí mismo con el paso del tiempo, pero si en relación a los logros de rendimiento del agente, decimos que es semidinámico.
- Discreto vs continuo: en relación al tiempo que dura la captación de percepciones y toma de acciones.
- Conocido vs desconocido: referido al conocimiento que el agente tiene acerca de las leyes físicas del entorno.

Estructura de los agentes

Internamente, decimos que un agente está formado por la unión de una arquitectura y un programa.

El trabajo de la IA es implementar programas de agente que utilicen la función de agente. Los programas de agente toman una percepción y devuelven una acción. El desafío consiste en generar agentes a partir de pequeños programas.

- Agente reflejo simple: Es el más sencillo. Toma decisiones en base a la percepción actual, ignorando el resto de las percepciones conseguidas. Se trata, por ejemplo, de programas escritos en base a reglas de condición-acción.
- Agente reflejo basado en modelos: Mantiene un estado interno que depende de la historia de percepciones, en otras palabras, guarda todo lo que sabe del mundo, incluso la parte que no ve en un determinado instante.
  Emplea dos tipos de conocimientos para actualizar este estado interno: la forma en que funciona el mundo, independientemente del agente, y la manera en que las acciones del agente afectan al ambiente.
- Agente basados en objetivos: Tener conocimiento acerca del estado actual del mundo, no siempre es suficiente para decidir qué hacer. Por lo general, el agente requiere información acerca del objetivo, es decir, aquello que se considera deseable o esperable. El agente debe ser capaz de combinar esta información con la que tiene del modelo, para elegir las acciones que conduzcan a ese objetivo. La búsqueda y el planeamiento son conceptos asociados al logro de los objetivos.
  Los distintos estados se clasifican en meta o no meta, el agente tiende a buscar los primeros. La representación del objetivo es explícita. Son agentes más flexibles, ya que pueden variar su destino si este es especificado como meta.
- Agente basados en utilidades: Los objetivos no son suficientes para generar comportamiento de "alta calidad" en la mayoría de los ambientes. Por ello, en ocasiones se asigna una utilidad a cada estado. La función de utilidad es una internalización de la medida de performance. Un agente racional maximiza la utilidad esperada.
- Agentes que aprenden: El ideal de la IA es construir agentes que aprendan, es decir, que sean capaces de operar sin conocimientos previos, y puedan volverse más competentes que al inicio.

AGENTES QUE RESUELVEN PROBLEMAS MEDIANTE BUSQUEDA.

Agentes que resuelven problemas:

Los agentes inteligentes buscan maximizar su medida de performance. Esto se simplifica si el agente encuentra un objetivo y trata de satisfacerlo. Para esto, se puede decir que se guía de tres pasos:

1. Formulación de meta (decidir qué estados son objetivo) y del problema (decidir qué acciones y estados se van a considerar).
2. Buscar una solución (examinar posibles acciones y armar una secuencia).
3. Ejecutar la solución.

Problemas y soluciones bien definidas:

Un problema puede ser definido formalmente mediante cinco componentes:

- Estado inicial, en el que el agente comienza.
- Una descripción de las posibles acciones disponibles para el agente. Función Actions(s)
- Una descripción de lo que hace cada acción, conocida como modelo de transición. El estado inicial, las acciones y el modelo de transición definen el espacio de estado del problema, representado mediante un grafo, en el que los nodos son estados y los vínculos, acciones.
- Comprobación de meta que determina si el estado actual es o no meta.
- Costo de camino, que es una función que asigna un costo numérico a cada ruta posible en el grafo.

Una solución es una secuencia de acciones que, aplicadas al estado inicial, me conducen a un estado meta. Una solución óptima es aquella en la que el costo de ruta es el mínimo de todas las soluciones posibles.

Formulando problemas:

Para formular problemas es obligatorio hacer abstracciones, es decir, remover detalles de una representación.
¿Qué abstraemos?

- Estados: se deben remover todos los detalles posibles conservando lo mínimo necesario para cumplir con el objetivo.
- Acciones: es preciso contemplar solo las acciones necesarias, expresadas de la manera más sencilla posible.

Buscando soluciones:

Un árbol de búsqueda es aquel en el que las aristas representan acciones y los nodos se relacionan a estados. El árbol presenta caminos, es necesario considerar todas las acciones posibles. Expandir el estado actual significa aplicar las acciones posibles y generar un nuevo conjunto de nodos. La estrategia de búsqueda determina qué nodo debe expandirse.
La frontera o lista abierta es la lista de nodos hoja que todavía no fueron procesados. En contraposición, el conjunto de nodos explorados o lista cerrada son todos aquellos estados que ya han sido procesados. El algoritmo que tiene en cuenta la lista cerrada se denomina búsqueda en grafo; el que no la considera, en cambio, es llamado búsqueda en árbol.

Infraestructura para algoritmos de búsqueda:

Los algoritmos de búsqueda requieren de una estructura de datos para mantener representaciones del árbol de búsqueda que está siendo construido. Por cada nodo n de un árbol, se tiene una estructura que contiene cuatro componentes:

- Estado: al que corresponde el nodo.
- Nodo padre: nodo que lo ha generado.
- Acción: que fue aplicada por el padre para generarlo.
- Costo de camino: denotado usualmente por g(n), es el costo desde el estado inicial hasta el nodo.

La frontera necesita almacenarse de una forma que permita al algoritmo elegir fácilmente el siguiente nodo a ser expandido. Por este motivo, la estructura adecuada es una lista que se comporta como cola, pila o cola priorizada. El conjunto de estados explorados puede ser implementado como una tabla hash.

Midiendo performance de los algoritmos:

Para poder seleccionar el algoritmo más correcto para la resolución de un problema, es posible evaluarlos mediante cuatro aspectos:

- Completitud: el algoritmo asegura encontrar una solución, si la hay.
- Optimalidad: el algoritmo encuentra una solución óptima.
- Complejidad temporal: ¿Cuánto tiempo toma encontrar la solución?
- Complejidad espacial: ¿Cuánta memoria necesito para encontrar la solución?

Las medidas de complejidad se van a expresar en términos de:

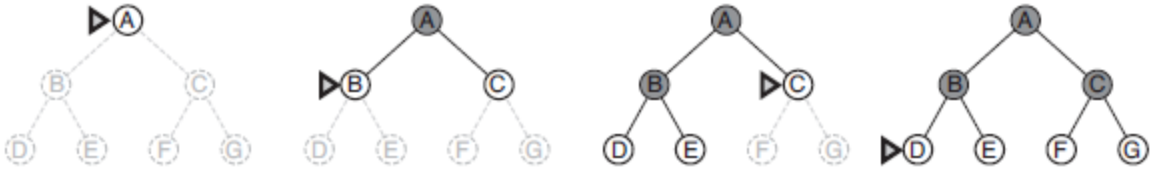
- El factor de ramificación, b.
- La menor profundidad de algún nodo meta, d.
- La longitud máxima de camino en el grafo de estados, m.

El tiempo se mide en términos de la cantidad de nodos generados; mientras que el espacio según la cantidad de nodos en memoria.

Estrategias de búsqueda sin información:

Búsqueda en amplitud:

Es una estrategia simple en la que se expande un nodo, luego sus sucesores, y luego los sucesores de estos. Utiliza una cola en la frontera.



El número total de nodos generados es: b + b^2 + b^3 + ... + b^d = O (b^d)

Búsqueda de costo uniforme:

Utiliza una cola priorizada en la frontera, ordenada por los costos de camino de los nodos. La función g(n) devuelve el costo del camino desde el nodo inicial al nodo actual.

Presenta dos diferencias significativas con respecto al algoritmo anterior:

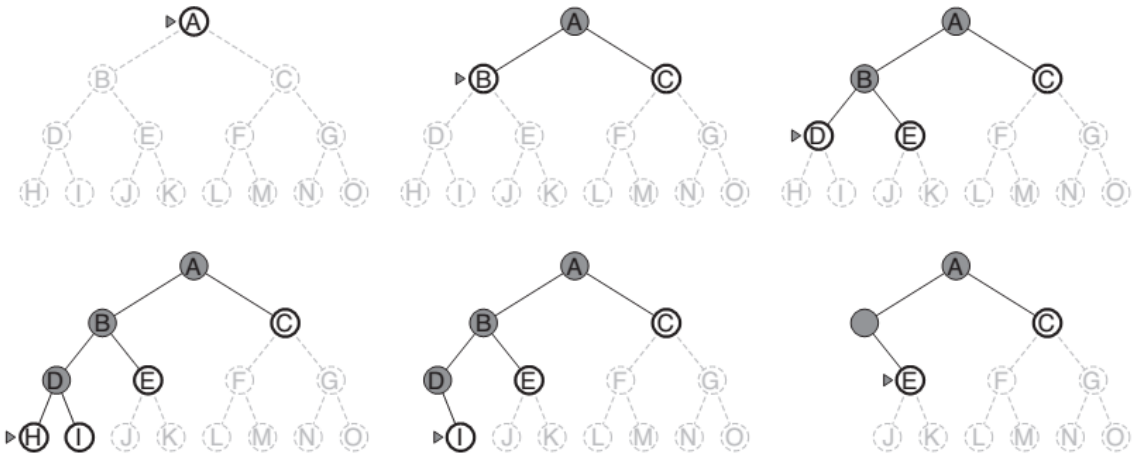
- La comprobación de meta se realiza recién al momento en que un nodo será expandido.
- Se agrega una chequeo por si se encuentra un estado ya generado pero de menor costo en frontera.

Búsqueda en profundidad:

Expande siempre el nodo de mayor nivel o profundidad en la frontera actual del árbol de búsqueda. Utiliza una pila en la frontera.

- La completitud depende del tipo búsqueda (en árbol o en grafo).
- No es óptima
- La complejidad temporal es O(b^m) (puede ser peor que O(b^d))
- La complejidad espacial es O(b*m)

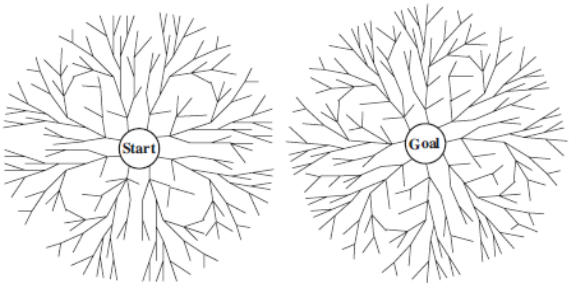
La variante backtracking search utiliza aún menos memoria.



Búsqueda en profundidad limitada:
Es similar a la búsqueda en profundidad, con la salvedad de que se limita la profundidad máxima a alcanzar. Introduce otro problema para la completitud, ya que el límite puede ser inferior a la profundidad de la mejor solución. Si conocemos el diámetro del espacio de estados, éste se puede usar como límite.

Búsqueda en profundidad iterativa:
Basado en las anteriores, encuentra el mejor límite aumentándolo gradualmente.

Búsqueda bidireccional:
La idea detrás de este algoritmo es correr dos búsquedas simultáneas, una desde el estado inicial y otra desde el estado meta.



- b (d/2) + b(d/2) << b^d
- Hay que tener acciones reversibles.
- Hay que conocer el estado meta.

Resumen de los algoritmos:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

MÉTODOS DE BÚSQUEDA CON INFORMACIÓN:

Estrategias de búsqueda informada:

Las búsquedas informadas, es decir, aquellas que usan conocimientos específicos del problema además de la definición del problema en sí mismo, pueden encontrar soluciones más eficientemente de lo que lo hacen las búsquedas no informadas.

El enfoque general que consideramos es llamado best-first search, una instancia del algoritmo de búsqueda en árbol o en grafo, que selecciona los nodos a expandir mediante una función de evaluación f(n), que le permite estimar costos. El nodo con el menor valor de evaluación es el que se expande primero. La elección de f determina la estrategia de búsqueda. Muchos algoritmos incluyen además una función heurística, que generalmente forma parte de f(n), y se denota con h(n). Esta función representa el costo estimado del camino más "económico" para alcanzar la meta desde un nodo n. Si n es un nodo meta, el valor de h(n) será 0.

A diferencia de g(n), h(n) depende únicamente del estado del nodo que toma como entrada, si bien ambas reciben como parámetro un nodo.

Búsqueda avara (greedy-search):

Trata de expandir el nodo que más se acerca a la meta, ya que persigue alcanzarla de la manera más rápida. Evalúa los nodos usando solo la función heurística, esto es f(n) = h(n).

- Si usamos búsqueda en árbol puede ser incompleta; si usamos búsqueda en grafo es completa en espacios no infinitos.
- No es óptima.
- La complejidad temporal y espacial en el peor de los casos es O(b^m); pero esto depende mucho de la heurística.
- Usa el mismo algoritmo que la búsqueda de costo uniforme, reemplazando g(n) por h(n).

Búsqueda A* (A estrella):

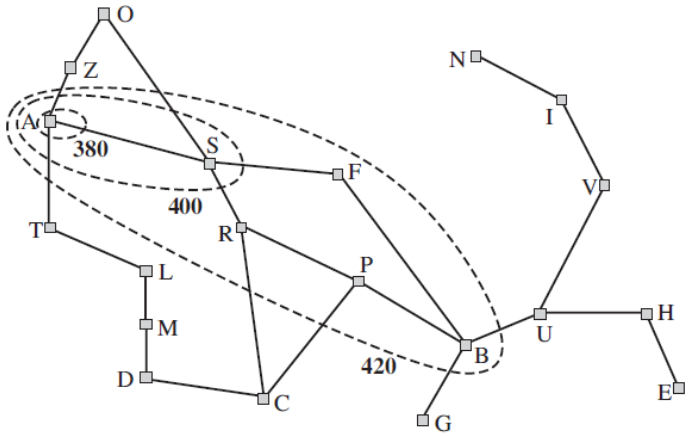
Minimiza el costo total estimado de la solución. Evalúa los nodos combinando g(n), costo de alcanzar el nodo, con h(n), costo del nodo a la meta: f(n) = h(n) + g(n).

- Si h(n) satisface determinadas condiciones, A* es completa y óptima. El hecho de que sea óptima implica que h(n) sea una heurística admisible y consistente.
- Entonces, presenta admisibilidad: nunca sobreestima el costo de llegar a la meta.
- Posee además consistencia (necesaria solo para búsqueda en grafo): para todo nodo n y todo sucesor de este, n', generado por una acción a, el costo estimado de alcanzar la meta desde n no es mayor al costo de paso de llegar a n más el costo estimado de alcanzar la meta desde n'. Es decir, h(n) <= c(n, a, n') + h(n').
- Usa el mismo algoritmo que la búsqueda de costo uniforme, reemplazando g(n) por g(n) + h(n).

Optimalidad de A*:

Dijimos que la búsqueda en árbol de A* es optima si h(n) es admisible, mientras que la búsqueda en grafo es óptima si h(n) es consistente.

A* busca en contornos de nodos del mismo costo total.



Si C* es el costo de la solución óptima:

- A* puede expandir algunos nodos dentro del contorno de la solución.
- A* expande todos los nodos con f(n) < C*.
- A* es óptimamente eficiente: no hay otro algoritmo que garantice expandir menos nodos que A*; asegurando la optimalidad de la solución.

Funciones heurísticas:

Una manera de caracterizar la calidad de una heurística es a través del factor de ramificación efectivo b*: N + 1 = 1 + b* + (b*)^2 + ... + (b*)^d *

Una heurística bien diseñada tiene b* cercano a 1. Si para cada nodo n, h2(n) >= h1(n) decimos que h2 domina a h1.

Generando heurísticas:

- Desde problemas relajados: el costo de una solución óptima de un problema relajado, es una heurística admisible para el problema original. Si se encuentran varias heurísticas alternativas, podemos generar una que domine a todas de la siguiente manera: h(n) = max{ h1(n), ... hm(n) }
- Desde subproblemas: el costo de solucionar un subproblema es menor que el de solucionar el problema completo. En las bases de patrones se almacenan los costos exactos para varios subproblemas, costos que luego son usados para calcular las heurísticas.

MÉTODOS DE BÚSQUEDA LOCALES.

Búsqueda local:

Los algoritmos de búsqueda local se utilizan cuando no importa el camino para llegar a la solución.

- Mantienen solo un nodo en memoria y se mueven a sus vecinos (usan poca memoria).
- Son útiles en problemas de optimización.
- Generalmente usan una formulación de estado completa.
- Una función objetivo determina que tan bueno es un estado. Es posible dibujar esta función respecto al espacio de estados para obtener un paisaje del espacio de búsqueda.

Un algoritmo es completo si siempre encuentra una solución, es óptimo si encuentra un máximo o mínimo global.

Ascenso de colina (Hill climbing):

- Siempre se mueve hacia arriba y termina cuando llega a un pico.
- Tiene problemas con máximos locales, crestas y mesetas.

Variantes:

- Stochastic hill climbing: elige al azar entre los movimientos ascendentes.
- First-choice hill climbing: genera los sucesores aleatoriamente de a uno, hasta que uno sea mejor que el actual (es útil cuando el factor de ramificación es alto).
- Random restart hill climbing: ejecuta varias iteraciones de hill climbing tradicional, comenzando aleatoriamente desde distintos estados iniciales.

Temple simulado (Simulated annealing):

- Combina el ascenso rápido con la aleatoriedad.
- A medida que transcurre el tiempo (o disminuye la temperatura) la posibilidad de elegir un sucesor peor disminuye.

Busqueda de haz local (Local beam search):

- Mantiene k nodos en memoria, en principio generados aleatoriamente.
- Por cada paso se generan todos los sucesores y se eligen lo k mejores.
- La información importante es compartida entre las distintas búsquedas paralelas.
- Los k estados pueden concentrarse rápidamente en un sector pequeño del espacio de estados.
- Aparece una variante llamada stochastic beam search: toma aleatoriamente los k sucesores, de acuerdo al valor de la función de cada uno.

Algoritmos genéticos:

- Es una variante de la búsqueda de haz local, solo que los sucesores se generan de a pares.
- Comienza con un conjunto llamado la población, compuesto de k estados o individuos.
- Se usa una función fitness, la cual devuelve valores mayores cuando es mejor un individuo.
- Se eligen k / 2 pares de individuos para ser reproducidos. La elección es al azar, con probabilidades proporcionales al fitness de cada individuo.
- Los pares son sometidos a una operación de crossover.
- Finalmente se pueden realizar mutaciones en los individuos resultantes.

Más búsqueda:

- Búsqueda con acciones no deterministas: las soluciones devuelven planes de contingencia en lugar secuencias de acciones.
- Búsqueda en ambientes parcialmente observables: los algoritmos trabajan con estados de creencia en lugar de estados reales.
- Búsqueda online: cuando el mundo o las acciones no son conocidos se necesita intercalar la búsqueda con la ejecución.

PROBLEMAS DE SATISFACCION DE RESTRICCIONES.

Un problema es considerado de satisfacción de restricciones cuando se lo representa con un conjunto de variables a las cuales se le tienen que asignar valores de manera de no violar las restricciones.

Definiendo problemas de satisfacción de restricciones:

Podemos afirmar que los CSP poseen 3 componentes, X, D y C:

- X es un conjunto de variables, {x1, x2, ..., xn}
- D es un conjunto de dominios, {d1, d2, ..., dn}, uno por variable. Cada dominio posee un conjunto de valores permitidos para la variable relacionada.
- C es un conjunto de restricciones que especifica las combinaciones permitidas de valores.

Una restricción consiste en una tupla <scope, rel>, donde scope es una lista de las variables afectadas y rel es la condición que se debe cumplir.
Cada estado en un CSP está definido por una asignación de valores a algunas o todas las variables. Una asignación que no viola ninguna de las restricciones es llamada asignación consistente o legal.
Una asignación completa es aquella en la que todas las variables tienen algún valor asignado. Una solución para CSP debe ser una asignación completa y consistente.

Ventajas de CSP sobre búsqueda tradicional:

- Posee una representación natural para una gran variedad de problemas.
- Son más rápidos ya que pueden eliminar grandes partes del árbol de búsqueda.
- En las búsquedas tradicionales solo podemos saber si un nodo es solución o no; en CSP podemos saber si una asignación parcial no conduce a una solución.
- En CSP podemos ver porqué una asignación no es solución.
- En CSP hay heurísticas genéricas para este tipo de problemas.

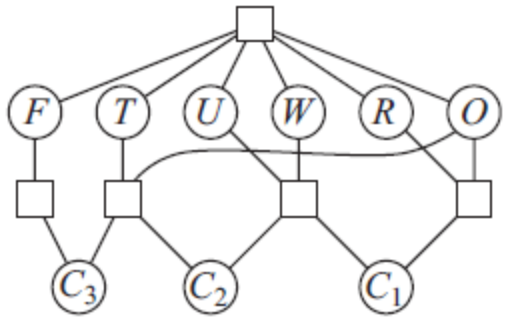
Variaciones sobre el formalismo CSP:

Los dominios pueden ser discretos y finitos, discretos e infinitos o continuos e infinitos.
Las restricciones pueden ser:

- Restricciones unarias: restringen el valor de una variable. Ej: SA <> blue
- Restricciones binarias: la restricción relaciona dos variables. Si un CSP solo posee este tipo de restricciones es un CSP binario. Ej: SA <> NSW o x < y
- Restricciones globales: involucran un número arbitrario de variables.

Cualquier CSP puede ser convertido a un CSP binario, aunque una restricción global puede ser más sencilla de leer y es posible diseñar inferencias de propósito especial que funcionen con las restricciones globales.

Representación mediante hipergrafo:



Los nodos (círculos) representan las variables y los hipernodos (cuadrados) representan las relaciones n-arias.

Propagación de restricciones:

La propagación de restricciones es un tipo de inferencia en la que se utilizan las restricciones para reducir el número de valores legales que puede tomar una variable, que a su vez puede disminuir el número de valores para la siguiente variable, y así sucesivamente.

La idea de la propagación es obtener lo que se conoce como consistencia local. Los diversos tipos de consistencia local son:

- Consistencia de nodo: Una variable es nodo-consistente si todos los valores de su dominio satisfacen las restricciones unarias. Una red es nodo-consistente si todas sus variables son nodo-consistentes.
- Consistencia de arco: Xi es arco-consistente respecto a Xj si, para cada valor de Di existe algún valor en Dj que satisfaga la restricción binaria en el arco (Xi, Xj). Una red es arco-consistente si cada variable es arco-consistente con las demás variables.
- Consistencia de camino: Un conjunto de 2 variables {Xi, Xj} es camino-consistente con respecto a una tercer variable Xm si, para cada asignación {Xi = a, Xj = b} consistente con las restricciones de {Xi, Xj}, existe una asignación a Xm que satisface las restricciones en {Xi, Xm} y {Xm, Xj}
- K-Consistencia: Un CSP es k-consistente si, para cualquier conjunto k - 1 de variables y para cualquier asignación consistente de esas variables, un valor consistente siempre puede ser asignado a cualquier k-ésima variable. A mayor valor de k, menor el tiempo de búsqueda de la solución; pero, la cantidad de tiempo y memoria utilizado para propagar la consistencia crece de manera exponencial a k.

Restricciones globales:

Son comunes en problemas reales y pueden ser manejadas por algoritmos especiales.

- Alldiff: comprueba que todos los valores sean distintos.
- Atmost: comprueba que como mucho se usen X recursos.

Backtraking search for CSP:

Backtracking trabaja con asignaciones incompletas. Respeta, al igual que todos los CSP, la propiedad de la conmutatividad, que enuncia que debe asignarse solo una variable por nodo.

- No se especifica estado inicial, action, result o goal (son siempre los mismos).
- La performance se puede mejorar mediante heurísticas genéricas.

Selección de variable:

- La siguiente.
- Minimun remaining values (MRV): elige la variable más restringida primero.
- Grado heurístico: selecciona la variable con mayores apariciones en restricciones primero.

Orden de valores del dominio:

- Least constraining value (LCV): elige primero el valor que menos opciones quita al resto de las variables.

Intercalando búsqueda e inferencia:

- Forward checking: cuando una variable X es asignada chequea toda variable conectada eliminando de sus dominios valores inconsistentes con la asignación de X. Se complementa bien con MRV ya que elimina valores y permite discernir las variables más restringidas.
- MAC (Maintaining Arc Consistency): llama a AC3 luego de cada asignación para aplicar consistencia de arco recursivamente.

Local search for CSP:

Usan asignaciones completas, inicialmente asignan un valor a cada variable y luego, en cada paso, eligen una variable para cambiar su valor.

- Para elegir el valor se usa la heurística mínimos conflictos.
- Se puede utilizar para mejorar soluciones cuando se producen cambios en el problema, por ejemplo, para problemas de planeación.

APRENDIZAJE DE MÁQUINA.

Aprendizaje basado en ejemplos:

En el que se describen los agentes que pueden mejorar su comportamiento a través del estudio diligente de sus propias experiencias. ¿Por qué queremos que un agente aprenda? Si es posible un mejor diseño, ¿porqué no lo diseñamos mejor desde el principio?

1. No siempre es posible anticipar todas las situaciones.
2. No se pueden anticipar todos los cambios.
3. El programador puede no tener en claro cómo programar una solución.

Componentes a aprender:

1. Mapeo de condiciones de un estado a acciones.
2. Inferir propiedades relevantes del mundo desde la secuencia de percepciones.
3. Como impactan las acciones.
4. Utilidad de los estados.
5. Información acerca de las preferencias de las acciones.

Representación del conocimiento:

- Representación factorizada como entradas (un vector de atributos).
- Aprendizaje inductivo es aprender una función general desde casos específicos.

Feedback:

Existen tres tipos de feedback:

1. Aprendizaje no supervisado: el agente aprende sin feedback. Lo más común son algoritmos de clustering.
2. Aprendizaje por refuerzo: el agente aprende a través de una serie de refuerzos (recompensas o castigos).
3. Aprendizaje supervisado: el agente cuenta con entradas y las salidas esperadas y aprende una función para realizar el mapeo.

El llamado aprendizaje semi-supervisado es un gris entre 1 y 3. Se debe al ruido o a la falta de etiquetas.

Aprendizaje supervisado:

Dado un Conjunto de entrenamiento (x1, y1), (x2, y2),..., (xn, yn) donde cada yj es generado mediante y = f(x), donde f es desconocida; se trata de encontrar un h(x) que aproxime a f.

- x e y pueden adoptar cualquier tipo de datos, xj es un vector de atributos
- h es una hipótesis. El aprendizaje consiste en buscar una hipótesis que se ajuste bien a los datos.

Se utiliza un conjunto de test para medir la precisión de una hipótesis. Decimos que una hipótesis generaliza bien, cuando predice correctamente con entradas no conocidas. Una hipótesis es consistente si se ajusta a todos los datos.
Cuando la salida es un valor de un conjunto finito, el problema es llamado clasificación. Cuando la salida es un número el problema es llamado regresión.
Existe un punto de equilibrio entre hipótesis complejas que se ajustan a los datos e hipótesis simples que generalizan mejor. Un problema es realizable si el espacio de hipótesis contiene a la función real. Hay un compromiso entre la expresividad del espacio de hipótesis y la complejidad de hallar una buena hipótesis.

Arboles de decisión:

Representan una función que toma como entradas un vector de atributos y devuelve un único valor. Las entradas y salida pueden ser discretas o continuas. Si solo existen dos posibilidades para la salida se denomina clasificación binaria. El árbol determina su decisión realizando una serie de tests. Cada nodo interno es un test sobre los valores de uno de los atributos, cada arista tiene las distintas opciones de la condición y cada nodo hoja es el valor que retorna la función.

Expresividad de los árboles de decisión:

Cualquier función en lógica proposicional se puede expresar como un árbol de decisión booleano
Goal <=> (path1 v path2 v ...)

- Algunas funciones se pueden representar más fácilmente que otras.
- Cada función es una tabla de verdad.
- Una tabla de verdad tiene 2 ^ n filas; donde cada una de estas filas puede tener dos valores para la misma.
- Nos quedan 2 ^ 2 ^ n tablas de verdad distintas.

Induciendo árboles de decisión:

Queremos encontrar un árbol de decisión que sea consistente con el conjunto de entrenamiento y lo más chico posible. Con algunas heurísticas se pueden encontrar árboles chicos (no el más chico).
DECISION-TREE-LEARNING usa un enfoque "divide y vencerás". Selecciona el atributo más importante y genera subarboles. Los ejemplos son cruciales para la construcción del árbol, pero los mismos no aparecen en el mismo. Se considera más importante a un atributo que hace la mayor diferencia de clasificación.

Midiendo la precisión del algoritmo. Curvas de aprendizaje:

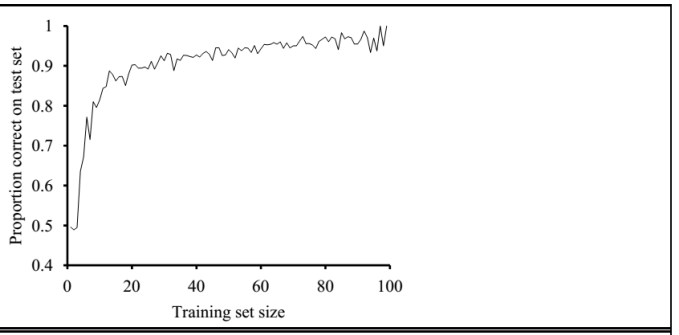


Figure 18.7 **FILES: .** A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Selección de atributos:

La idea es elegir un atributo que mejor clasifique los ejemplos.

- Atributo perfecto: divide los ejemplos en conjuntos donde todos los ejemplos son positivos o negativos.
- Atributo inútil: divide en conjuntos donde los ejemplos se clasifican en igual proporción. Patrons no es perfecto pero es bastante bueno.

Una medida formal se puede obtener mediante la noción de ganancia de información, la cual se define en términos de entropía, que es la medida de incertidumbre de una variable aleatoria. La ganancia de información reduce la entropía. Una moneda que cae siempre de cara, tiene entropía cero.

Generalización y sobreentrenamiento:

El sobreentrenamiento es un fenómeno que se da en cualquier algoritmo de aprendizaje. Se produce cuando el modelo se adapta perfectamente a los patrones de entrada pero es incapaz de generalizar. Las posibles soluciones son:

- Poda del árbol: elimina test irrelevantes.
- Early stop: Corta la ejecución del algoritmo cuando quedan solo atributos irrelevantes.

Este último implica menos trabajo pero no permite reconocer situaciones donde la combinación de atributos es relevante. (XOR)

Ampliando la aplicabilidad de los AD:

Se necesita poder manejar estos aspectos para poder aplicarlos en aplicaciones reales:

- Datos faltantes.
- Atributos multivaluados: problemas con la ganancia de información.
- Atributos enteros o continuos: puntos de división.
- Atributos de salida continuos: árboles de regresión.

Los AD son generalmente el primer método que se trata de aplicar a un dataset. Una propiedad importante de los AD es que son comprensibles por un ser humano.

Seleccionando la mejor hipótesis:

Queremos una hipótesis que se comporte bien con datos futuros. Cuando hablamos de datos futuros, asumsión de estacionaridad. Al referirnos a "bien", consecuentemente definimos "error rate" como la proporción de errores que se comete.

- Holdout cross-validation: separa los datos en training y test. No se aprende de todos los ejemplos.
- k-fold cross-validation: separa los datos en k subconjuntos. Se hacen k entrenamientos, siempre tomando 1 conjunto para test.
- leave-one-out cross-validation: k = 1

Los datos se entrenan con el conjunto de training; se elige la mejor hipótesis con los datos de test y se informa la tasa de error del conjunto de validación.

Selección del modelo:

La selección de la mejor hipótesis se divide en dos tareas:

- Elegir el espacio de estados.
- Encontrar la mejor hipótesis dentro del espacio.

Una manera es parametrizar los modelos de acuerdo al tamaño. La selección del modelo penalizando las hipótesis complejas es llamada regularización.

From error rate to loss:

Función loss: define la pérdida de utilidad debido a un error.
L(x, y, y') = Utilidad(x, y) - Utilidad(x, y')

- L1(y, y') = |y - y'|
- L2(y, y') = (y - y')^2
- L0/1 (y, y´) = 0 if y == y' else 1

Regresión y clasificación con modelos lineales:

El espacio de hipótesis está formado por funciones lineales de entradas con valores continuos. Se conoce como regresión a la tarea de encontrar un h que minimice la pérdida empírica, tradicionalmente se usa L2.

Regresión lineal univariada:

- Tiene la forma de una recta: y = w1 x + w0
- Usamos w porque vamos a pensar los coeficientes como pesos (weights).
- w es el vector definido por [w0, w1]
- hw(x) = w1 x + w0

Método hill-climbing con descenso por el gradiente:

$\mathbf{w} \leftarrow$ any point in the parameter space

loop until convergence **do**

  for each w_i **in w do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

Donde alfa es llamado "learning rate". Puede ser constante o variable reduciéndose a medida que transcurre el aprendizaje.

- Descenso por el gradiente en lote (batch): utiliza las fórmulas de actualización que contemplan todos los ejemplos.
- Descenso por el gradiente estocástico: se aplica la regla de actualización individual eligiendo un ejemplo al azar.

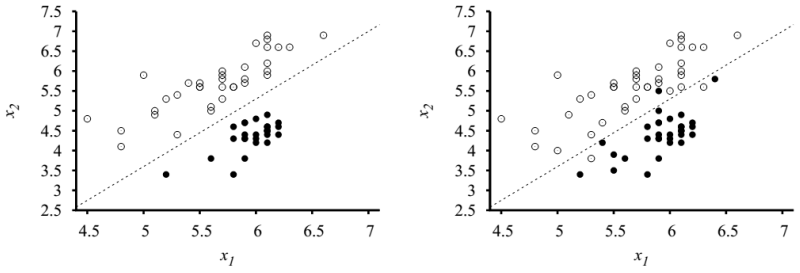
Regresión lineal multivariada:

Es la misma idea pero x es un vector. Si creamos una entrada xj, 0 que siempre valga 1 podemos expresar h como el producto punto o producto de matrices.
La regla de actualización es

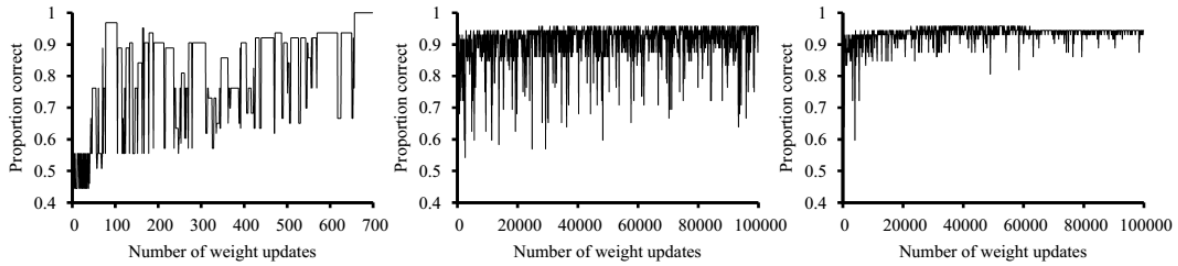
$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}(y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \ .$$

Con regresión multivariada es común usar regularización para evitar overfitting.

Clasificadores lineales con umbral:



- Una frontera de decisión es una línea que separa dos clases.
- Una frontera lineal se llama separador lineal y los datos que los admiten linealmente separables.
- La función threshold no es derivable, por lo tanto no se puede hacer descenso por el gradiente.
- En su lugar podemos usar la regla de aprendizaje del perceptrón.



Clasificadores lineales con regresión logística:

La función threshold tiene algunos inconvenientes:

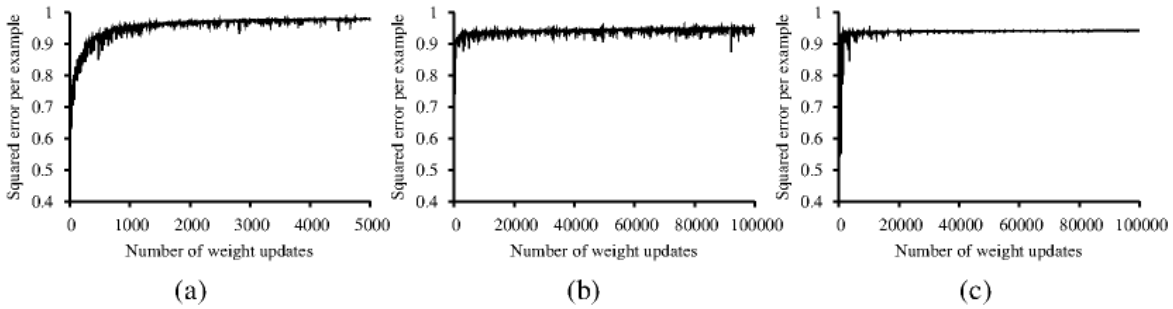
- No es derivable, lo cual hace que el aprendizaje sea poco predecible.
- Valores muy cercanos a la frontera son clasificados con 0 o 1.

La función logística solventa estos problemas.

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}} .$$

La regla de actualización es:

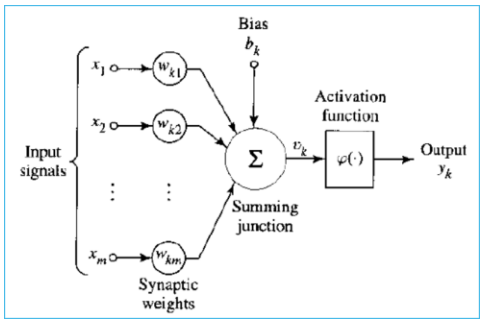
$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$



Redes neuronales artificiales:

Una red neuronal es una gran cantidad de unidades simples de cómputo interconectadas entre sí, que tratan de asemejarse al cerebro en dos aspectos:

1. El conocimiento es adquirido por la red desde el ambiente a través de un proceso de aprendizaje.
2. El conocimiento adquirido es almacenado en las conexiones entre neuronas.



Estructura de una red neuronal:

- Una red está compuesta por unidades (o neuronas) conectadas por links.
- Un link tiene asociado un peso y es usado para propagar la activación desde una neurona a otra.
- La activación de una neurona aj se calcula:

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

Donde:

- g es la función de activación.
- w es el peso sináptico.
- ai es la entrada relacionada.

Si g es la función threshold, la neurona se denomina perceptron. Hay redes propagación hacia adelante y recurrentes. Las redes feed-forward se pueden organizar en capas o layers. Si la red es multicapa, las capas que no se conectan con la salida se denominan ocultas.

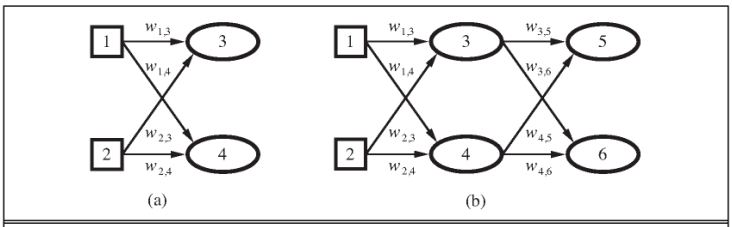


Figure 18.20 (a) A perceptron network with two inputs and two output units. (b) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights.

Redes de una capa:

Si hay m salidas, entonces hay m redes distintas, cada una con su propio entrenamiento. Cada neurona es un clasificador lineal.

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Redes multicapa. MLP:

Mediante este tipo de redes es posible representar cualquier tipo de función, incluso funciones no lineales. Las salidas de una capa son entradas a otra capa.

Entrenamiento del MLP: back propagation:

El valor de las salidas depende de todos los pesos de entrada. Otro inconveniente es que el error en las salidas debe retro propagarse a la capa oculta. La idea básica es dividirlo de acuerdo a los pesos en la capa oculta.

Datos adicionales:

Existen otros modelos de red además del MLP:

- Radial Base Networks
- Self Organized Maps (SOM)
- Hopfield

En problemas reales es necesaria una etapa de preprocesamiento.

Modelos no paramétricos:

- Modelo paramétrico: conjunto fijo de parámetros.
- Modelo no-paramétrico: la cantidad de parámetros es ilimitada; cada ejemplo es un parámetro. Se lo conoce también como aprendizaje basado en instancia o aprendizaje basado en memoria. Una posible implementación es la tabla de búsqueda.

Modelo de los vecinos cercanos (KNN):

El resultado se basa en los k vecinos más cercanos.

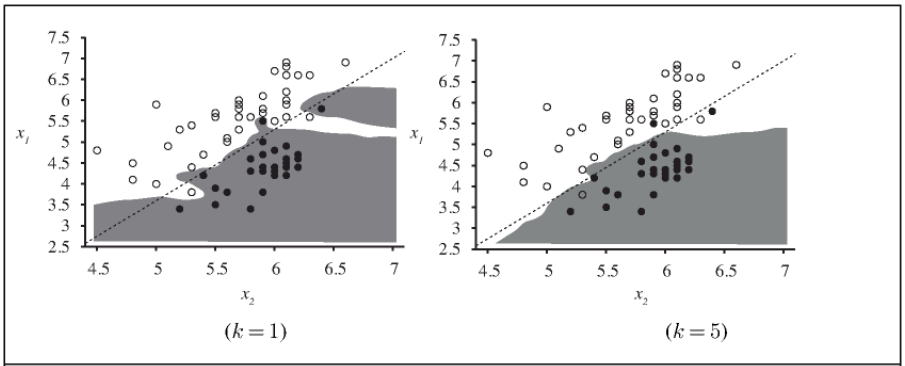


Figure 18.26 (a) A k-nearest-neighbor model showing the extent of the explosion class for the data in Figure 18.15, with $k = 1$. Overfitting is apparent. (b) With $k = 5$, the overfitting problem goes away for this data set.

¿Cómo medir la distancia? Distancia de Minkowski.

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = (\sum_i |x_{j,i} - x_{q,i}|^p)^{1/p}$$

Si p=2 se conoce como distancia Euclidea y si p=1 como distancia de Manhattan.
Es importante la normalización de los datos. Una posibilidad es (Xi - media) / desvio; otra es (Xi - min) / (max - min). Los algoritmos knn funcionan bien con muchos ejemplos y pocas dimensiones.
Variantes para mejorar la velocidad:

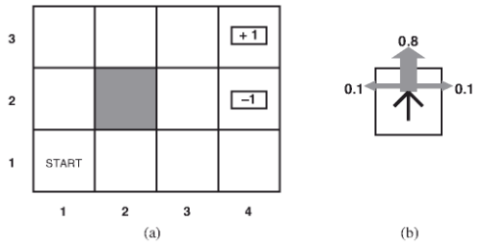
- k-d tree
- Locality- sensitive hashing

Otros temas:

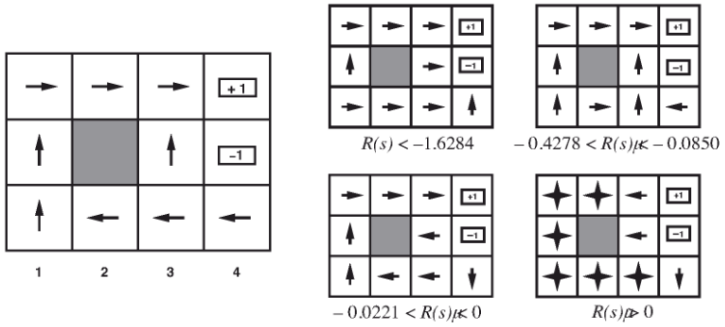
- Support vector machines: es el método más popular para cuando no se conoce nada del problema.
- Ensemble learning: combina k hipótesis para tratar de reducir el error.

Markov Decision Process:

- Ambiente secuencial, completamente observable, estocástico.
- Modelo de transición Markoviano (la probabilidad de alcanzar s2 desde s1 depende solo de s1 y no de la historia de estados).
- Las recompensas son aditivas (la utilidad es la suma de las recompensas).



Una solución a un MDP es una política π .
$\pi(s)$es la acción recomendada por la política en el estado s.
- La calidad de una política se mide en términos de la utilidad esperada
- Una política óptima π^* maximiza la utilidad esperada.



Utilidades en el tiempo:

Las recompensas pueden ser:

- Aditivas:
  $U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots$.
- Depreciativas:
  $U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$

Donde γ es el factor de descuento, un valor entre 0 y 1.

Políticas óptimas y utilidad de los estados:

La utilidad esperada ejecutando en un estado inicial s es:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

Una política óptima estaría dada por:

$$\pi_s^* = \underset{\pi}{\operatorname{argmax}}\, U^{\pi}(s)$$

El agente selecciona la acción en base al principio de maximización de la utilidad esperada:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) U(s')$$

Algoritmos para encontrar políticas óptimas

- Value iteration
- Policy iteration

Ambos métodos hacen pequeños ajustes por cada iteración para calcular las utilidades de cada estado. Ambos necesitan conocer el modelo de transición y la función R(s).

Aprendizaje por refuerzo:

La tarea del aprendizaje por refuerzo es usar los refuerzos recibidos para aprender una política óptima para el ambi ente.
Vamos a considerar tres diseños de agente:

- Agente basado en utilidad: aprende una función de utilidad basada en los estados. Necesita conocer un modelo del ambiente
- Agente Q-learning: aprende una función (Q-function) de utilidad para las acciones disponibles. No necesita modelo del ambiente.
- Agente reflejo: aprende una política que mapea estados en acciones.

Dos posibles tipos de aprendizaje:

- Aprendizaje pasivo: tiene una política fija, solo aprendemos las utilidades.
- Aprendizaje activo: el agente tiene que aprender que hacer.

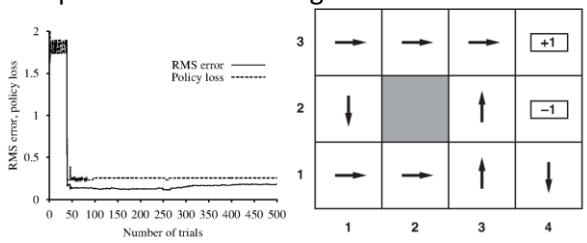
Hay además varios métodos de resolución, como la programación dinámica, los métodos de Montecarlo, el aprendizaje por diferencia de tiempo (TD), etc.

Aprendizaje activo:

El agente no posee una política fija, sino que debe aprenderla. Pero, cuando el agente encuentra que determinado comportamiento es bueno, ¿debe siempre ejecutarlo? Surge la distinción entre explotación y exploración.

Exploración vs explotación:

Comportamiento de un agente avaro:



Un agente debiera hacer un balance entre explotar los conocimientos que posee para maximizar las recompensas y explorar para mejorar los conocimientos y maximizar la utilidad esperada a largo plazo.

Función de exploración:

Dadas las acciones disponibles, devuelve qué acción tomar dependiendo de las utilidades de las mismas y las frecuencias de aparición. Debe seguir el principio GLIE (Greedy in the Limit of Infinite Exploration).

Aprendiendo Q-Functions:

Q-learning: aprende utilidades por acción. No necesita modelo del ambiente.
Un algoritmo similar a Q-Learning es SARSA (State-Action-Reward-State-Action).
Solo cambia la regla de actualización:
Regla de Q-Learning:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma max_{a'} Q(s',a') - Q(s,a))$$

Regla de SARSA:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$