Apuntes resumidos Inteligencia A...

user120723

Inteligencia Artificial

3º Grado en Ingeniería Informática

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

INTELIGENCIA ARTIFICIAL

Con LiiNCE, lo único que te va a costar es estudiar

-80% de descuento en tus gafas graduadas

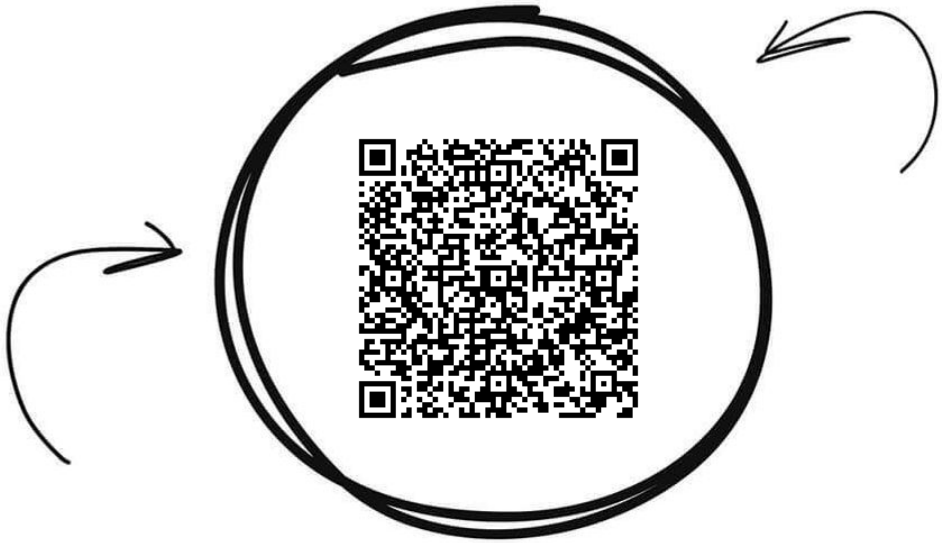
Gafas graduadas desde 15€

Pide cita online en **liince.com**

Esquina de San Ignacio de Loyola con Lacarra de Miguel

LiiNCE

Inteligencia Artificial



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

1. AGENTES DE RESOLUCIÓN DE PROBLEMAS Y BÚSQUEDA NO INFORMADA

1.1. AGENTES DE RESOLUCIÓN DE PROBLEMAS

Un **agente racional** es una entidad que percibe y actúa para conseguir unos objetivos. De forma abstracta es una función de las percepciones recogidas a las acciones. La **percepción** y **actuación** dependen del entorno en el que esté situado.

Hay tres tipos de agentes:

- **Agente reflejo**: ignora la historia y no considera las consecuencias de sus acciones.
- **Agente reflejo basado en modelo**: es dependiente de la historia.
- **Agente basado en objetivo y modelo**: las decisiones se basan en las consecuencias de sus acciones, un modelo de cómo evoluciona el mundo con sus acciones, y describe situaciones deseables (objetivo).

Los agentes de resolución de problemas son agentes basados en objetivos. Se describe un proceso que elige y organiza acciones anticipando cuál será su resultado y las acciones van dirigidas a la consecución de un objetivo.

Para automatizar la resolución de problemas se deben considerar tres aspectos:

- **Modelos conceptuales** (o matemáticos) para formalizar la clase de problemas (modelo de estados, modelos probabilísticos,etc.)
- **Lenguaje de representación** atómica, factorizada, estructurada,etc.
- **Algoritmos** para resolver problemas representados por los modelos.

Los pasos a seguir para resolver problemas son:

- Formulación del objetivo: definir los estados del mundo que queremos alcanzar.
- Formulación del problema: definir los estados y acciones a considerar para alcanzar el objetivo.
- Búsqueda: determinar la posible secuencia de acciones que nos llevan al objetivo.
- Ejecución: realizar las acciones de la secuencia.

Los agentes de resolución de problemas utilizan **representaciones atómicas** (cada estado tiene un símbolo). El modelo informa al algoritmo de si dos **estados** son **iguales** o no, de las **acciones** aplicables a un estado, de los **estados sucesores** y de si un **estado** es **objetivo**.

No se puede representar todo el espacio de estados en memoria. Por lo que el modelo del sistema va generando el espacio de estados durante la exploración.

1.2. DEFINICIÓN DEL PROBLEMA

El **espacio de estados** del problema está definido por: estado inicial + acciones + modelo de transiciones. Forma un **grafo dirigido** en el que los nodos son estados y los enlaces son acciones. Un **camino** es una secuencia de estados conectados por una secuencia de acciones. La **solución** es la secuencia de acciones que lleva desde el estado inicial al estado final.

La **definición de un problema** se basa en definir para dicho problema lo siguiente:

- Estados.
- Estado inicial.
- Acciones aplicables.
- Modelo de transiciones.
- Test objetivo.
- Coste camino.

El **modelo de transiciones** debe representar las **precondiciones** y sus **efectos**. El **coste del camino** es el sumatorio de pasos.

1.3. ALGORITMOS DE EXPLORACIÓN

La búsqueda simula la exploración del espacio de estados, generando los sucesores de los estados ya explorados (**expandir estados**). La búsqueda genera un **árbol de búsqueda** partiendo de un nodo raíz (estado inicial) y genera otros nodos (estados que apuntan al padre y a los hijos). Al conjunto de nodos sin hijos (no expandidos) se le denomina **frontera**. Se puede llegar a un estado por **múltiples caminos** y volver al mismo estado después de visitarlo.

El **grafo del espacio de estados** es una representación matemática del problema de búsqueda:

- Los **nodos** son configuraciones del mundo.
- Los **arcos** representan transiciones de estado como resultado de las acciones.
- El **test de objetivo** es un conjunto de uno o más nodos.

En un grafo de estados, cada estado aparece solo una vez. Normalmente no se puede representar un grafo entero en memoria (muy grande).

DESCRIPCIÓN INFORMAL DE LA BÚSQUEDA EN ÁRBOL

function BÚSQUEDA-ÁRBOL (*problema*, *estrategia*) **returns** **solución** o **fallo**
  Inicializa la *frontera* utilizando el estado inicial del problema
  loop do
    if la *frontera* está vacía **then return** *fallo*
    elige un *nodo* hoja de la frontera de acuerdo a una *estrategia*
    if el *nodo* contiene un nodo objetivo **then return** *solución*
    expande el nodo elegido, añadiendo los nodos resultantes a la frontera

La desventaja de realizar la búsqueda en árbol es que puede repetirse infinitamente la misma estructura.

DESCRIPCIÓN INFORMAL DE LA BÚSQUEDA EN GRAFO

function BÚSQUEDA-GRAFO(***problema***, ***estrategia***) **returns** **solución** o **fallo**
Inicializa la ***frontera*** utilizando el estado inicial del problema
Inicializa el conjunto de nodos **explorados** a vacío
loop do
if la ***frontera*** está vacía **then return** ***fallo***
elige un ***nodo*** hoja de la frontera de acuerdo a una ***estrategia***
if el ***nodo*** contiene un nodo objetivo **then return** ***solución***
añade el ***nodo*** al conjunto de nodos ***explorados***[2]
expande el nodo elegido, añadiendo los nodos resultantes a la
frontera sólo si no están en la ***frontera*** o en el conjunto ***explorados***

ESTRATEGIAS DE BÚSQUEDA

Una estrategia de búsqueda viene definida por la elección del **orden de expansión** de nodos. Las estrategias se **evalúan** según estas 4 dimensiones:

- **Completitud**: ¿Encuentra una solución si existe?
- **Optimalidad**: ¿Encuentra siempre la solución de menor coste?
- **Complejidad temporal**: nodos generados/expandidos.
- **Complejidad espacial**: nº de nodos almacenados en memoria en la búsqueda.

La complejidad temporal y espacial se miden en términos de la dificultad del problema:

- **b**: factor de ramificación máximo (nº máximo de sucesores de un nodo)
- **d**: profundidad de la solución de menor coste.
- **m**: máxima profundidad de cualquier camino en espacio de estados (puede ser infinita)

La **búsqueda ciega** utiliza sólo información disponible en la definición del problema.

BÚSQUEDA PRIMERO EN ANCHURA (BFS)

Primero expande el nodo no expandido **menos profundo**. La frontera es una cola **FIFO**.
El **test** lo aplica al nodo cuando es **generado**. Es **completa** si es nodo objetivo menos profundo está en una profundidad finita **d** y si **b** es finito. Es **óptima** si el coste del camino es una función no decreciente de la profundidad.

La **complejidad temporal** es $O(b^{d+1})$, asumiendo que cada nodo tiene b sucesores y que la solución tiene una profundidad d, ya que en el peor de los casos se generan todos los nodos del último nivel y se expanden todos menos el último nodo de profundidad d.

La **complejidad espacial** es $O(b^{d})$. Si es búsqueda en grafo, la complejidad espacial viene dada por el tamaño de nodos de la frontera y es en árbol, la complejidad espacial sería la misma, pero se perdería mucho tiempo por los caminos redundantes

Dado que se trata de una **complejidad exponencial**, los requisitos de memoria suponen un problema.

BÚSQUEDA COSTE UNIFORME (UC)

Es una extensión de búsqueda en anchura de forma que se modifica el algoritmo para que sea óptimo para cualquier función de coste en cada paso. Se expande el nodo con **menor coste**. La frontera es una cola **ordenada por coste de caminos**. La BFS y UC son iguales cuando todos los costes de las operaciones son iguales. El **test** se aplica cuando es **expandido**.

La **completitud** se garantiza si cada paso tiene al menos un coste mayor igual a un pequeño valor constante positivo ε. Si es completa es **óptima**.

La **complejidad temporal** si se asume C* coste de la solución óptima y que cada acción cuesta al menos ε es de $O(b^{1+(C^*/\varepsilon)})$, igual que la **complejidad espacial**.

BÚSQUEDA PRIMERO EN PROFUNDIDAD (DFS)

Expande primero el nodo **más profundo**. La frontera es una cola **LIFO**.

Si la búsqueda es en **grafo** es **completa** en estados finitos. Sin embargo, si es en **árbol** es **no completa** porque puede haber bucles infinitos. **Ambas** son **no óptimas**.

La **complejidad temporal** en la búsqueda en **grafo** está acotada por el tamaño del espacio de estados, mientras que en la búsqueda en **árbol**, dado que se pueden generar todos los nodos del árbol, es de $O(b^m)$, siendo m la **máxima profundidad** de cualquier nodo.

En cuanto a **complejidad espacial**, para una búsqueda en grafo no hay ventaja, pero para una búsqueda en **árbol** solo necesita almacenar un único camino de la raíz al nodo hoja, junto a la frontera $O(bm + 1)$

BÚSQUEDA PROFUNDIDAD LIMITADA (DLS)

Los nodos en la profundidad **l** no tienen sucesores, por lo que se resuelve el problema de los caminos infinitos. Si **l<d** es **incompleto**. Si **l>d** es **no óptimo**. La complejidad temporal es de $O(b^l)$ y la espacial de $O(bl)$.

BÚSQUEDA ITERATIVA (IDS)

Intenta **combinar** el comportamiento **espacial** de la búsqueda en **profundidad** con la **optimalidad** de la búsqueda en **anchura**. El algoritmo consiste en realizar búsquedas en profundidad sucesivas con un nivel de profundidad máximo acotado y creciente en cada iteración. Además, esto permite evitar los casos en que la búsqueda en profundidad no acaba. En la primera iteración, la profundidad máxima será 1 y este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución. Para garantizar que el algoritmo acaba se puede definir una cota máxima de profundidad.

Es **completa** y **óptima**. Su **complejidad temporal** es de $O(b^d)$ y su **complejidad espacial** de $O(bd)$.

BÚSQUEDA BIDIRECCIONAL

Se realizan búsquedas simultáneas desde el objetivo y el inicio. Comprueba si el nodo pertenece a la otra frontera antes de expandir. La **complejidad espacial** es el punto débil. Es **completa** y **óptima** si ambas búsquedas son en **anchura**.

RESUMEN DE ALGORITMOS

Criterio	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Completitud	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
Temporal	b^{d+1}	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Espacial	b^{d+1}	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimalidad	YES*	YES*	NO	NO	YES	YES

2. BÚSQUEDA INFORMADA

Las búsquedas informadas emplean conocimientos del problema.

2.1. BÚSQUEDA PRIMERO EL MEJOR

La idea es utilizar una **función de evaluación** de cada nodo **f(n)** que es una estimación del coste, por lo que se elegirá el nodo con menor coste estimado. La elección de f(n) estima la **estrategia**.

La mayoría de los algoritmos primero el mejor incluyen una función **heurística h(n)**, que es una estimación del coste del camino menos costoso desde el estado en el nodo n al objetivo. **h(n)** es sólo función del estado y es una función no negativa, específica del problema y con **h(n)=0** si el estado del nodo es un **objetivo**.

El algoritmo es idéntico a coste uniforme, salvo que la estrategia que emplea es f(n)=h(n) en lugar de g(n), siendo g(n) el coste para llegar al nodo objetivo desde n.

2.2. ALGORITMO VORAZ / GREEDY ALGORITHM

El algoritmo voraz expande el nodo que parece estar más **próximo al objetivo**. Es un algoritmo **no óptimo** y solo es **completo** si la búsqueda se hace en grafo y el espacio es finito. La **complejidad temporal** es de $O(b^m)$, igual que la **complejidad espacial**. Una buena heurística puede reducir la complejidad.

2.3. BÚSQUEDA A*

La idea es evitar expandir nodos que tienen caminos costosos. **f(n) = g(n) + h(n)**, siendo **g(n)** el coste real para alcanzar el nodo n, **h(n)** la estimación del coste para alcanzar el objetivo desde el nodo n, y **f(n)** la estimación del coste total del camino desde n al objetivo. El algoritmo es idéntico a coste uniforme, empleando f(n) en lugar de g(n).

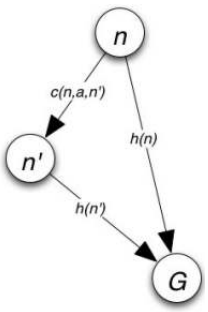
Es una búsqueda **óptima** y **completa** si la función heurística es **admisible en árbol** y es **consistente en grafo**.

Una heurística es **admisible** si nunca sobreestima el coste real de alcanzar el objetivo. Una heurística admisible es optimista. Formalmente:

- **h(n) <=h*(n)** donde h*(n) es el coste real para llegar al objetivo desde n.
- **h(n)>=0**, de forma que **h(G)=0** para cualquier objetivo.

Una heurística es consistente si **h(n)<=c(n,a,n')+h(n')**.

La **complejidad temporal** es exponencial con la longitud del camino. Respecto a la **complejidad espacial**, se almacenan todos los nodos.



2.4. HEURÍSTICAS MÁS INFORMADAS

Dadas dos heurísticas A* h_1 y h_2, se dice que h_2 está **mejor informada** que h_1 si para cualquier estado n del espacio de búsqueda $\mathbf{h_1(n)<=h_2(n)}$.

2.5. MEDIDA DE CALIDAD DE LAS HEURÍSTICAS Y RENDIMIENTO

$$\text{Factor de ramificación medio real} = \frac{\text{total-estados-generados}}{\text{nodos-expandidos}}$$

$$\text{Factor ramificación medio árbol generado} = \frac{\text{total-nodos-generados}}{\text{nodos-expandidos}}$$

b* es el **factor de ramificación efectivo** y mide la calidad de la heurística. Si el A* ha generado N nodos, es el factor de ramificación que un árbol uniforme de profundidad d tendría para contener N+1 nodos.

$$0 = \frac{b^*(1-(b^*)^d)}{1-b^*} - N$$

La ecuación se resuelve con el método de la bisección:

Sea $[a_0,b_0]$ nuestro intervalo
El error al tomar el punto medio como solución es error ≤ (b-a)/2

MIENTRAS QUE error > error_admitido
  Llamemos m al punto medio de a0 y b0 =m=(b0-a0)/2

  Si f(m)=0, casualmente hemos encontrado: SOL = m. PARAMOS

  Si f(m)≠0 elegimos entre
    [a0,m] si es que f(a0)·f(m)<0, haciendo [a1,b1]= [a0,m]
    [m,b0] si es que f(m)·f(b0)<0, haciendo [a1,b1]= [m,b0]

    Calculo de nuevo el error con el nuevo intervalo
FIN MIENTRAS QUE

SOL=punto medio el último intervalo considerado

2.6. BÚSQUEDA HEURÍSTICA CON MEMORIA ACOTADA

Dado que la búsqueda A* tiene limitaciones de espacio, se pueden implementar otras búsquedas en su lugar.

ITERATIVE-DEEPENING A* (IDA*)

Igual que IDS pero con límite dado por f en cada iteración, es decir, en lugar de limitar la profundidad, se limita **f** al mínimo coste de cualquier nodo descartado en la iteración anterior. De esta forma, si se genera un nodo cuyo coste excede el límite en curso se descarta. El algoritmo expande nodos en orden creciente de coste, por lo que el primer objetivo es **óptimo**. La **complejidad temporal** es de $O(b^d)$.

BÚSQUEDA PRIMERO EL MEJOR RECURSIVO (RBFS)

Mantiene el valor de **f de la mejor alternativa** disponible de cualquier ancestro del nodo. Si el valor f en curso excede el valor alternativo, se vuelve al camino alternativo. Al hacer **backtracking** se cambia f de cada nodo en el camino por el mejor valor de f en sus hijos. RBFS **recuerda el mejor valor de f** de las hojas de sub-árboles olvidados y puede decidir si merece la pena volver a re-expandirlos.

Es un poco más eficiente que IDA*. Es **óptima** si h(n) es admisible. La **complejidad temporal** es difícil de caracterizar, pero la **complejidad espacial** es de $O(bd)$.

(SIMPLIFIED) MEMORY BOUNDED A* (S)MA*

Expande la mejor hoja hasta que se llena la memoria. Cuando se llena, SMA* elimina el peor nodo y, como RBFS, devuelve el valor del nodo olvidado a su padre. Es **completa** y **óptima** si la solución es alcanzable.

3. BÚSQUEDA LOCAL

3.1. PROBLEMAS DE OPTIMIZACIÓN Y BÚSQUEDA LOCAL

Los algoritmos de búsqueda local trabajan utilizando **un único nodo en curso** en lugar de una frontera y generalmente se mueven solo a nodos vecinos. Algunas de las ventajas que presentan es que usan **poca memoria** y encuentran una **solución razonable** en **espacios de estados grandes** o infinitos. Son útiles para problemas de optimización puros ya que encuentran el estado mejor de acuerdo a una **función objetivo**.

3.2. ESCALADA / HILL-CLIMBING SEARCH

"Es un bucle que se mueve continuamente en la dirección del valor creciente". Termina cuando se alcanza un **"pico"** sin vecinos con mejor valor. Hill-climbing no mira más allá de los vecinos inmediatos al estado en curso. Esta búsqueda elige aleatoriamente entre el conjunto de sucesores que maximiza/minimiza la función objetivo si hay más de uno. También se le conoce como **búsqueda local voraz** (greedy local search).

El algoritmo es el siguiente:

```
function HILL-CLIMBING( problema) return un estado que es un máximo local
enCurso← MAKE-NODE(INITIAL-STATE[problema])
loop do
    vecino← sucesor de enCurso con mayor valor
    if vecino.VALUE ≤ enCurso.VALUE then return enCurso.STATE
    enCurso← vecino
```

Hay métodos aleatorios en escalada:

- **Escalada estocástica**: elige un movimiento aleatorio con probabilidad de selección en función de la pendiente. Converge más lentamente, pero puede encontrar mejores soluciones.
- **Primera elección en escalada**: genera aleatoriamente sucesores hasta que se encuentra uno mejor que el nodo en curso. Es una buena opción cuando hay miles de sucesores.
- **Reinicio aleatorio**: si fracasa, reintenta con otro estado inicial aleatorio, evitando así atascos en un máximo local. Es una búsqueda completa ya que, con infinitos intentos, la probabilidad de generar el estado objetivo es 1. Si cada escalada tiene una probabilidad de éxito p, el número de reinicios es 1/p.

3.3. ENFRIAMIENTO SIMULADO / SIMULATED ANNEALING

Permite escapar de los máximos locales ya que permite "malos" movimientos. La idea es reducir gradualmente el tamaño del salto para salir del máximo local y su frecuencia. Si la "temperatura" decrece lo suficientemente despacio, alcanzaremos el mejor estado.

Pseudo-código del Algoritmo Simulated Annealing

```
function Simulated_Annealing(T0, k, Tfinal)
T <- T0
Sactual <- Solución inicial
for T=T0 to Tfinal do
   if T = 0 then return Sactual
   else Snuevo <- Nueva solución aleatoria
   Inc(C) <- Coste(Snuevo)-Coste(Sactual)
   if (Inc(C) > 0) then Sactual <- Snuevo
   else Sactual <- Snuevo con probabilidad Exp(Inc(C) /F(T))
                    donde F(T) = k Exp(-delta*T)
    T <- Evolucion(T)
```

donde, Inc(C) = [Coste(nuevo estado) − Coste(estado anterior)]

- Cuando Inc(C) < 0:la probabilidad de cambio tiene una P [aceptación] = 1
- Cuando Inc(C) > 0: la probabilidad de cambio tiene una P[aceptación] = $\exp(\frac{Inc(c)}{F(T)})$

donde F (T) = **k** · exp(−**δ T**)

El algoritmo necesita de los siguientes **parámetros**:

- **T**: el paso de iteración en la ejecución.
- **k**: velocidad que tarda en comenzar a decrecer la temperatura.
- **δ**: velocidad a la que desciende la temperatura.

3.4. BÚSQUEDA LOCAL EN HAZ / LOCAL BEAM SEARCH

En esta búsqueda se mantienen **k estados** en lugar de uno que, inicialmente, se generan aleatoriamente. Se determinan los mejores sucesores de los k estados y, en caso de que cualquiera de los sucesores sea el objetivo, se acaba. En caso contrario, se seleccionan los k mejores sucesores y se repite el proceso. Dado que puede darse el caso de que los k estados acaben en el mismo máximo local, existe una **variante estocástica** que elige los k sucesores con una probabilidad proporcional a una función de éxito (**selección natural**).

La búsqueda en haz local es una generalización de la escalada (High Climbing = Local Beam Search (k=1). Abriendo la ventana de sucesores elegibles, mejoran las posibilidades de escapar de mesetas formadas por la función heurística y de encontrar caminos más cortos hasta alguna meta. Sin embargo, no siempre el algoritmo encuentra soluciones mejores con valores de k mayores.

3.5. ALGORITMOS GENÉTICOS

Los **estados** se representan como **strings de dígitos** y se realiza una **clasificación de la población** por la función de **fitness**. En esta versión, la probabilidad de **elección** de los individuos para la reproducción es directamente proporcional a la función fitness. Finalmente se realiza la **reproducción** en la que se elige un punto de mezcla aleatorio.

```
function GENETIC_ALGORITHM( población, FITNESS-FN) return un individuo
  input: población, un conjunto de individuos
         FITNESS-FN, una fución que determina la calidad del individuo
  repeat
         nueva_población ← conjunto vacio
         for i :=1  to SIZE(población) do
                  x ← RANDOM_SELECTION(población, FITNESS_FN)
                  y ← RANDOM_SELECTION(población, FITNESS_FN)
                  hijo← REPRODUCE(x,y)
                  if (pequeña probabilidad aleatoria) then hijo← MUTATE(hijo)
                  add hijo to nueva_población
         población  ← nueva_poblaciónn
  until algún individuo encaje suficientemente o haya pasado suficiente tiempo
  return el mejor individuo
```

```
function REPRODUCE( X, Y) returns un individuo
  input: x,y, individuos padres

   n ← LENGTH(x);
   c ← número aleatorio de 1 a n
   return  APPEND (SUBSTRING(x,1,c), SUBSTRING(y,c+1,n))
```

4. JUEGOS

4.1. JUEGOS

Los juegos son una forma de entorno **multiagente**. La solución es una estrategia que especifica el **movimiento a cada réplica del oponente**. La **limitación en tiempo** fuerza a una **solución aproximada**. Hay una **función de evaluación** que evalúa lo buena que es una posición del juego.

En los juegos en los que hay **dos agentes** estos tienen funcionalidad opuesta. Si la solución es un único valor, un agente pretende **maximizarlo** y otro **minimizarlo**.

En los juegos que hay **más de dos agentes** se ven como **economías**, en los que cada agente tiene funciones de utilidad distinta.

Los tipos de juegos son:

	Determinista	Azar
Información perfecta	Ajedrez, damas, othello	backgamon, monopoly
Información imperfecta		Bridge, póquer, scrabble

Los juegos más típicos en IA son de dos jugadores: **MAX** Y **MIN**. MAX mueve primero y alterna juegos con MIN hasta que el juego acaba. Los juegos **suma-zero** de información perfecta son deterministas, completamente observables y sus valores de utilidad al final del juego son siempre opuestos siendo su suma es siempre igual (gana +1, pierde -1, empate 0).

Los juegos como búsqueda tienen los siguientes atributos:

- **Estado inicial**.
- **Jugador**(s): define el jugador que mueve en un estado.
- **Acciones**(s): movimientos legales en un estado
- **Resultado**(s,a): **modelo de transición** que define el resultado de un movimiento.
- **Test-terminal**(s): **estado terminal** si el juego está finalizado.
- **Función utilidad**: valor numérico del estado terminal.
- **Estrategia o política**: S → A

MAX utiliza un **árbol de juego** para determinar el siguiente movimiento (los nodos representan los estados del juego y los arcos los movimientos legales). **Podar** el árbol nos permite ignorar partes del árbol de búsqueda. Las **funciones de evaluación heurística** nos permiten aproximar la utilidad real de un estado sin hacer una búsqueda completa.

Se denomina **árbol de búsqueda** al árbol superpuesto al árbol de juego completo y que examina suficientes nodos para permitir decidir al jugador que movimiento realizar.

4.2. DECISIONES ÓPTIMAS

Dado un árbol de juego, la estrategia óptima puede ser determinada por el valor **minimax** de cada nodo.

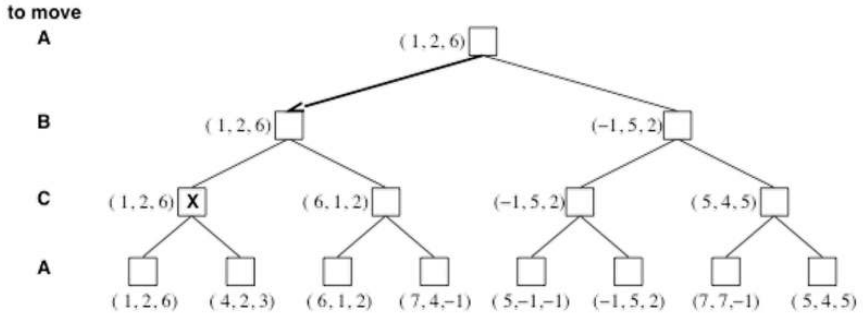
VALOR-MINIMAX(s)=

UTILIDAD(s)	Si Test-terminal (s)
$\max_{a \in Acciones(s)}$ **MINIMAX**(Resultado(s,a))	Si *Jugado(s)=MAX*
$\min_{a \in Acciones(s)}$ **MINIMAX**(Resultado(s,a))	Si *Jugador(s)=MIN*

El algoritmo minimax es el siguiente:

```
function MINIMAX-DECISION(estado) returns una acción
  v←MAX-VALOR(estado)
  return la acción en Resultados(estado, acción) con valor v
function MAX-VALOR(estado) returns un valor utilidad
  if TEST-TERMINAL (estado) then return UTILIDAD(estado)
  v ← - ∞
  for each a in Acciones(estado) do
    v ← MAX(v, MIN-VALOR(Resultado(s,a)))
  return v
function MIN-VALOR(estado) returns un valor utilidad
  if TEST-TERMINAL (estado) then return UTILIDAD(estado)
  v ← ∞
  for each a in Acciones(estado) do
    v ← MIN(v, MAX-VALOR(Restultado(s,a)))
  return v
```

En juegos con **más de dos jugadores** los valores **minimax** se convierten en **vectores**. El nodo devuelto al nodo n es siempre el vector del estado sucesor con el valor más alto del jugador eligiendo en n.



El algoritmo **minimax** es **completo** y **óptimo**. Su **complejidad temporal** es $O(b^m)$ y la **complejidad espacial** $O(bm)$.

4.3. PODA α-β

Dado que el número de estados del juego es exponencial con el número de movimientos, se puede realizar una **poda alfa-beta** de forma que no sea necesario examinar cada nodo:

- **Alfa**: valor de la mejor elección encontrada en cualquier punto del camino MAX.
- **Beta**: valor de la mejor elección encontrada en cualquier punto del camino MIN.

De esta forma si en un modo min V<=α, max poda esa rama. Y si en un nodo max V>=β, min poda esa rama. Al comenzar la búsqueda el **rango** es [-∞,+∞]. El **algoritmo** es:

```
function ALPHA-BETA-SEARCH(estado) returns una acción

  v←MAX-VALOR(estado, - ∞ , +∞)
  return la acción en  ACCIONES(estado) con valor v

-----------------------------------------------------

function MAX-VALOR(estado, α , β) returns un valor utilidad
  if  TEST-TERMINAL (estado) then return UTILIDAD(estado)
  v ← - ∞
  for each a in ACCIONES(estado) do
    v ← MAX(v, MIN-VALOR(RESULTADO(s,a), α , β))
    if v ≥ β then return β
    α ← MAX(α ,v)
  return v

function MIN-VALOR(estado, α , β) returns un valor utilidad
  if TEST-TERMINAL(estado) then return UTILIDAD(estado)
  v ← + ∞
  for each a in ACCIONES(estado) do
    v ← MIN(v, MAX-VALOR(RESULTADO(s,a), α , β))
    if v ≤ α then return α
    β ← MIN(β ,v)
  return v
```

La poda no afecta a los resultados finales y se pueden podar subárboles completos.

El orden de los movimientos puede mejorar la efectividad de la poda y con una "ordenación perfecta", la **complejidad temporal** sería de $O(b^{m/2})$, lo que implica una **reducción del factor de ramificación efectivo**. Esto sugiere que merece la pena examinar los sucesores que son probablemente mejores primero. Los mejores movimientos son **killer moves**.

Hay que **evitar estados repetidos**, por lo que hay dos opciones para ello:

- Almacenar los movimientos que dan lugar al mismo estado en memoria (**transposition moves**).
- Almacenar los movimientos en una tabla hash (**transposition table**).

4.4. JUEGOS CON INFORMACIÓN IMPERFECTA

Dado que minimax y la poda alfa-beta requieren evaluaciones de demasiados nodos, puede ser impracticable en una suma de tiempo razonable. Por ello se propuso cortar la búsqueda antes, reemplazando **test-terminal** por **test-corte**, y aplicar una **función de evaluación heurística EVAL**, reemplazando la función de utilidad de alfa-beta. De esta forma se introduce un **límite de profundidad fijo** o **límite en el tiempo de exploración**. Cuando ocurre el corte, se realiza la evaluación.

La **función heurística EVAL** produce una estimación de la utilidad esperada del juego para una posición dada. Las prestaciones dependen de la calidad de EVAL. Los requisitos son:

- EVAL debería ordenar los nodos terminales de la misma forma que UTILITY.
- El cómputo de la función no debe ser costoso.
- Para estados no terminales, EVAL debería estar fuertemente correlacionada con las oportunidades reales de ganar.

La mayoría de las funciones de evaluación utilizan varias **características**. El conjunto de estas define **clases de equivalencia de estados**.

La evaluación debería ser aplicada únicamente a **posiciones estables** (**quiescent**), de manera que las posiciones no estables deben extenderse más hasta alcanzar una estable. La búsqueda extra se denomina **quiescent search**.

También se puede realizar una **poda hacia delante**, en la que sólo se consideran n mejores movimientos.

4.5. JUEGOS DE AZAR

En este tipo de juegos no se puede calcular valores definidos de **minimax**, solo **valores probables** (esperados), que son **valores promedio en los nodos**. El minimax pasa a ser **expectimax**.

EXPECTED-MINIMAX (s)=

UTILIDAD(s)	Si *TEST-TERMINAL(s)*
$\max_a$ EXPECTED-MINIMAX(RESULT(s,a))	SI *JUGADOR(s)* es MAX
$\min_a$ EXPECTED-MINIMAX(RESULT(s,a))	Si *JUGADOR(s)* es MIN
$\sum_r P(r)$. EXPECTED-MINIMAX(RESULT(s,a))	Si *JUGADOR(s)* es AZAR

donde r representa valores del dado (o probabilidad del evento)

La **complejidad temporal** es de $O(b^m * n^m)$, donde n es el número de posibilidades.

5. REPRESENTACIÓN DEL CONOCIMIENTO. REGLAS

5.1. REPRESENTACIÓN DEL CONOCIMIENTO. REGLAS

Los **problemas de planificación** son problemas de resolución que se representan de forma explícita con las representaciones de estado y acción. La búsqueda en el espacio de estados puede progresar de dos formas:

- Razonando progresivamente: estado inicial al objetivo
- Razonando regresivamente: del objetivo al estado inicial.

Se quiere reconocer patrones en los estados para desencadenar acciones, por lo que se usan **sistemas basados en reglas**.

SISTEMAS DE PRODUCCIÓN / LENGUAJES BASADOS EN REGLAS

Un **sistema de producción** ofrece un mecanismo de control dirigido por patrones para la resolución de problemas.

Las herramientas para implementar búsquedas en el espacio de estados son:

- **Representación del estado del sistema**: (deftemplate estado (slot garrafa (type INTEGER)))
- **Estado inicial**: (estado (garrafa 0))
- **Estado final**: (estado (garrafa 3))
- **Operadores**: (defrule Agnade_Un_Litro
  ?estado <- (estado(garrafa ?cantidad))
  =>
  (modify ?estado (garrafa (+ ?cantidad 1))))

Una regla no aporta mucho, pero un **conjunto de reglas** pueden formar una cadena capaz de alcanzar una conclusión significativa.

Al utilizar un lenguaje basado en reglas se busca simular el razonamiento humano en dominios en los que el conocimiento es "evolutivo" y en los cuales no existe un método determinista seguro (finanzas, medicina, ciencias sociales, etc.). El **método** a seguir es:

1. Aislar y modelar un subconjunto del mundo real
2. Modelar el problema en términos de hechos iniciales o de objetivos a conseguir
3. Modelar los patrones y las acciones simulando el razonamiento humano.

ARQUITECTURA DE LOS LENGUAJES BASADOS EN REGLAS

La **base de hechos** contiene hechos iniciales, más los deducidos. La **base de reglas** contiene las reglas que explotan los hechos. El **motor de inferencia** aplica las reglas a los hechos. Un sistema basado en reglas tiene **dos partes**:

- **Parte declarativa**:
  - Hechos: conocimiento declarativo (información) sobre el mundo real.
  - Reglas: conocimiento declarativo de la gestión de la base de hechos.
    - En lógica *monótona* solo se permite añadir hechos
    - En lógica *non-monótona* se permiten inserciones, modificaciones y eliminaciones.
  - Meta-reglas: conocimiento declarativo sobre el empleo de las reglas.
- **Parte algorítmica/imperativa**:
  - Motor de inferencia: software que efectúa los razonamientos sobre el conocimiento declarativo disponible.

La **interpretación de las reglas** conlleva los siguientes pasos:

1. **Reconocimiento**: comparación de los patrones en las reglas con los elementos de la memoria de trabajo.
2. **Resolución de conflictos**: se elige una regla entre las satisfechas por la memoria de trabajo. La **estrategia de control** especifica la forma de resolver los conflictos.
3. **Ejecución**: se ejecuta su parte ENTONCES. La ejecución da lugar a cambios en la memoria de trabajo.

REPRESENTACIÓN DE HECHOS Y REGLAS

En **CLIPS** los **hechos** se representan así:
- Pepe nació en Zaragoza en 1996: (Persona (Nombre Pepe) (Edad 30) (Trabajo Ninguno))

Las **reglas** se representan así:

- **Antecedentes** que reconocen estructuras de símbolos.
- **Consecuentes** que contienen operadores especiales que manipulan las estructuras de símbolos

```
(defrule Pepe-desempleado
   (Persona (Nombre Pepe) (Edad 30) (Trabajo Ninguno))
=>(assert (Tarea (Reclamar Paro) (Para Pepe))))
(defrule Inscripción-paro
   (Tarea (Reclamar Paro) (Para Pepe))
=>(assert (Inscrito_Paro (Nombre Pepe))))
```

PROCESO DE RECONOCIMIENTO

Existen dos **estructuras y organizaciones básicas**:

- **Redes de inferencia**: las conclusiones de las reglas son hechos que se corresponden exactamente con premisas de otras reglas.
- **Sistema de reconocimiento de patrones**: los patrones descritos en las premisas de las reglas son más generales y pueden reconocer distintos hechos.

PATRONES CON VARIABLES

- variable := ?<nombre>

Se puede ligar un hecho a una variable también:

```
(deftemplate persona (slot nombre) (slot direccion))
(deftemplate cambio (slot nombre) (slot direccion))
(defrule procesa-informacion-cambios
   ?h1 <- (cambio (nombre ?nombre) (direccion ?direccion))
   ?h2 <- (persona (nombre ?nombre))
=>
   (retract ?h1)
   (modify ?h2 (direccion ?direccion))
```

?: variable que no liga valor y reconoce cualquier valor.
$?: comodín que reconoce 0 o más valores de un atributo multivalor
Operador ~: negación
Operador |: OR
Operador &: En combinación con los anteriores para ligar valor a una variable
La función **bind** liga un valor no obtenido mediante reconocimiento a una variable:

```
(bind ?suma (+ ?a ?b))
```

5.2. META-REGLAS. CONTROL EN SISTEMAS DE PRODUCCIÓN

Para controlar el sistema de producción se toman 3 aproximaciones:

- **Insertar control en las reglas**: cada grupo de reglas tienen un patrón que indica en qué fase se aplican. Una regla de cada fase se aplica cuando no quedan más reglas aplicables.
- **Utilizar prioridades**: no se garantiza el orden de ejecución correcto.
- Insertar patrón de fase en reglas de conocimiento y separar reglas de control que cambian de fase: se definen reglas de control como las siguientes:

```
(deffacts informacion-control
  (fase deteccion)
  (secuencia-fases aislar recuperaracion deteccion))

(defrule cambia-fase
  (declare (salience -10))
  ?fase <- (fase ?fase-en-curso)
  ?lista <- (secuencia-fases ?siguiente $?resto)
=>
  (retract ?fase ?lista)
  (assert (fase ?siguiente))
  (assert (secuencia-fases ?resto ?siguiente)))
```

Hay que **evitar utilizar prioridades**, para ello en las propias reglas se puede poner como condición que no se cumpla una que tiene mayor prioridad.

```
(defrule A-Ganar
  ?f <- (elige-movimiento)
  (cuadro-abierto gana)
=>
  (retract f)
  (assert (mueve gana)))
```

```
(defrule A-Bloquear
  ?f <- (elige-movimiento)
  (cuadro-abierto bloquea)
  (not (cuadro-abierto gana))
=>
  (retract f)
  (assert (mueve bloquea)))
```

```
(defrule cualquiera
  ?f <- (elige-movimiento)
  (cuadro-abierto
        ?c&iz|der|medio)
  (not (cuadro-abierto gana))
  (not (cuadro-abierto bloquea)
=>
  (retract f)
  (assert (mueve-a ?c)))
```

En CLIPS se pueden definir **módulos**: (defmodule <nombre-modulo> [<comentario>])
El módulo en curso se cambia cuando se especifica el nombre de un módulo en la construcción o mediante (set-current-module <nombre-modulo>)

deftemplate (y todos los hechos que lo usan) pueden ser compartidos con otros módulos mediante **export** o **import**. Una construcción debe definirse antes de importarse, pero no tiene que estar definida antes de exportarse. El único módulo que puede **redefinir** lo que importa y exporta es MAIN.

Cada módulo tiene su propia agenda y CLIPS mantiene un current focus que determina la agenda que se usa. El current focus por defecto es MAIN. La instrucción (focus <modulo>) cambia el current focus y apila el anterior en una pila (**focus stack**). Si una regla tiene declarado **auto-focus** con el valor true, se activa su módulo automáticamente si la regla se sensibiliza.

5.3. EFICIENCIA. SISTEMAS DE RECONOCIMIENTO DE PATRONES

Los tipos de información a retener son:

- Número de condiciones satisfechas
- Mantenimiento de la memoria (α-memorias)
- Relación de condiciones (β-memorias)
- Mantenimiento del conjunto conflicto

La red de **RETE** y su algoritmo de propagación aprovechan la **redundancia temporal** (α-memorias,β-memorias y agenda) y la **similitud estructural** (si se repite la misma restricción en distintas reglas se aprovecha la misma rama). Las reglas se compilan bajo la forma de dos tipos de árbol:

- **Árbol de discriminación**: filtrado y propagación de los nuevos hechos en memoria de trabajo. Cada hoja tiene una premisa y cada nodo una comprobación (test).
- **Árbol de consistencia**: verifica las correspondencias entre variables. Cada nodo reagrupa 2 a 3 las premisas de una misma regla

Para escribir **programas eficientes** hay que seguir los siguientes criterios:

- Los patrones más específicos primero
- Colocar al principio patrones que reconocen pocos hechos
- Patrones que reconocen hechos "volátiles" al final

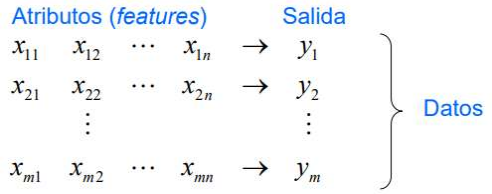
El **test** en los patrones debe ir lo más al principio posible.

6. INTRODUCCIÓN AL APRENDIZAJE

6.2. APRENDIZAJE

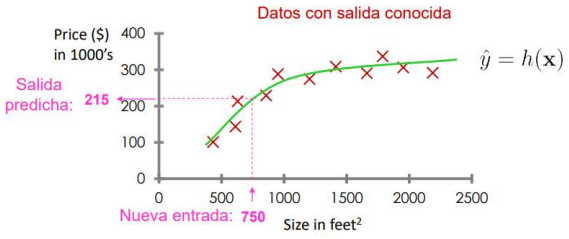
El **aprendizaje** es un conjunto de métodos capaces de descubrir automáticamente patrones en datos y usarlos para predecir datos futuros o para tomar decisiones bajo incertidumbre (Kevin P. Murphy 2012). Dado un conjunto de datos X, el objetivo del aprendizaje es encontrarle sentido a dicho conjunto. Hay tres **tipos de problemas**: supervisado, no supervisado y refuerzo.

En el **aprendizaje supervisado**, para cada elemento del conjunto X se conoce un valor asociado de salida, Y. La salida es generada por una función desconocida y=f(x).

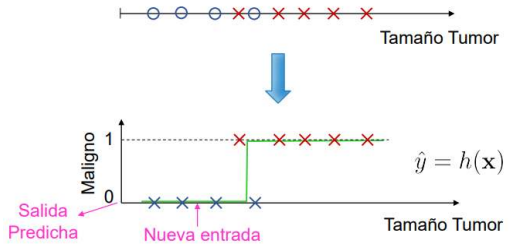


Hay que descubrir una función h que aproxime la función real f y que permita predecir la salida para datos futuros y=h(x).

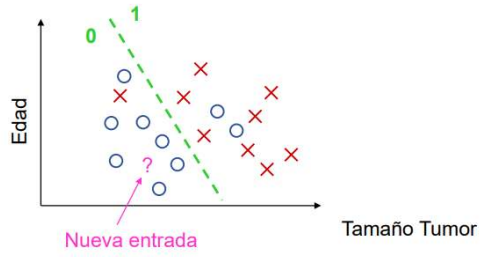
La **regresión** genera una salida continua.



La **clasificación** genera una salida discreta, que puede ser binaria o multi-clase.



Se puede dar también una **clasificación con varios atributos**.



Al **aprendizaje no supervisado** también se le llama "*knowledge discovery*". Como datos de entrenamiento se usan solo las entradas, de forma que el objetivo es descubrir un **patrón** en los datos. El **clustering** se basa en descubrir agrupaciones en los datos. ¿Cuántos hay? ¿A qué cluster pertenece cada datos?

El **aprendizaje por refuerzo** se basa en aprender la política de comportamiento. El algoritmo aprende en base a experiencias pasadas propias o ajenas (expertos).
Se asocian recompensas a las buenas acciones.

El **proceso de aprendizaje** se divide en:

- **Fase de entrenamiento**: se utilizan los datos de partida X para aprender el modelo de interés.
- **Fase de validación**: Utilizar un subconjunto diferente dentro de X para "medir" la calidad del modelo.
- **Fase de test**: Evaluar el modelo en nuevos datos desconocidos

6.2. CLASIFICADOR BASADO EN PROBABILIDADES

Representa la **incertidumbre** y tiene un **fundamento matemático sólido**.

Una pequeño recordatorio de probabilidades:

- Probabilidad **condicional**: $P(x \mid y) \equiv \frac{P(x,y)}{P(y)}$ *P. conjunta*
- Tma Probabilidad **Total**: $P(x) = \sum_y P(x,y) = \sum_y P(x \mid y)P(y)$
- *Regla de Bayes*: $P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)} = \frac{P(x \mid y)P(y)}{\sum_{y'} P(x \mid y')P(y')}$
  P. conjunta / *P. individual*
- X e Y son **independientes** si: $X \perp Y$

$$\forall x, y : P(x,y) = P(x)P(y)$$
$$P(x \mid y) = P(x)$$
$$P(y \mid x) = P(y)$$

Saber Y no influye en X

En cuanto a la **probabilidad conjunta**, para N variables binarias, hacen falta 2^N-1 valores para especificar la distribución conjunta. La **probabilidad marginal** puede obtenerse a partir de la conjunta, mediante la marginalización: $\mathbf{P(x) = \Sigma_y P(x,y)}$.

La **independencia condicional** se denota mediante X⊥Y|Z y se lee: X e Y son condicionalmente independientes dado Z. Se comprueba de la siguiente manera:

$$\forall x, y, z : P(x, y \mid z) = P(x \mid z)P(y \mid z)$$
$$P(x \mid y, z) = P(x \mid z)$$
$$P(y \mid x, z) = P(y \mid z)$$

Conocido Z, Y no da información adicional sobre X, ni X sobre Y

6.3. TIPOS DE ERRORES EN CLASIFICACIÓN Y MÉTRICAS

Los tipos de **errores** son:

- **Verdadero positivo (TP)**: dato positivo correctamente clasificado como positivo.
- **Verdadero negativo (TN)**: dato negativamente correctamente clasificado como negativo.
- **Falso positivo /Error tipo I (FP)**: dato negativo incorrectamente clasificado como positivo.
- **Falso negativo / Error tipo II (FN)**: dato positivo incorrectamente clasificado como negativo.

Las **métricas** para la evaluación de clasificadores son:

- **Accuracy**: $\frac{TP+TN}{TP+FP+TN+FN}$
- **Precision**: fracción de los clasificados como positivos/negativos que son en realidad positivos/negativos. Un clasificador con alta/baja precisión devuelve muchos/pocos resultados, pero la mayoría de ellos son/no son correctos.

  $\frac{TP}{TP+FP} \in [0,1] \qquad \left(\frac{TN}{TN+FN}\right)$
- **Recall**: fracción de los positivos/negativos que hemos clasificado correctamente como positivos/negativos. Un clasificador con recall alta/baja devuelve la mayor/menor parte de los resultados correctos.

  $\frac{TP}{TP+FN} \in [0,1] \qquad \left(\frac{TN}{TN+FP}\right)$
- **Curva precision-recall**: gráfico bidimensional que resulta de unir los pares precision-recall en diferentes experimentos de clasificación cambiando el umbral de clasificación τ. Cambiando el umbral se puede llevar al clasificador a operar en distintos modos de funcionamiento. El **área bajo la curva** (AUC) se suele tomar como métrica para evaluar clasificadores.

  $\tau \uparrow \Rightarrow FP \downarrow \Rightarrow FN \uparrow \Rightarrow P \rightarrow 1 \Rightarrow R \rightarrow 0$

  $\tau \downarrow \Rightarrow FP \uparrow \Rightarrow FN \downarrow \Rightarrow P \rightarrow 0 \Rightarrow R \rightarrow 1$
- **F1 score**: media armónica entre precisión y recall.
  $F_1=(1+ \beta^2)*(precision*recall)/(\beta^2*precision+recall)$
- **Matriz de confusión**:tabla en la que las filas se refieren a las categorías reales y las columnas a las categorías predichas y en cada celda se almacena la cuenta o la frecuencia de cada caso. Las diagonales indican las clasificaciones correctas.

7. REDES BAYESIANAS

Una **red bayesiana** es un **modelo probabilístico gráfico** (grafo dirigido **acíclico**) que representa un conjunto de variables aleatorias y sus dependencias condicionales. Los **nodos** son las variables (con sus dominios) y los **arcos** codifican la dependencia condicional (suelen representar relación causal, pero no es necesario).

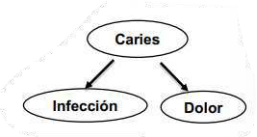
Existe una **distribución** condicional **por cada nodo**, es decir, una colección de distribuciones sobre X, una por cada combinación de los valores de los nodos padre. Las probabilidades se representan en una **tabla** de probabilidades condicionales.

Una red Bayesiana representa **implícitamente** las **distribuciones conjuntas**. En la tabla de cada nodo se representan las probabilidades locales. Las distribuciones conjuntas son el producto de las distribuciones condicionales locales.

$$P(x_1, x_2, \cdots x_n) = \prod_{i=1}^{n} P(x_i \mid padres(X_i))$$

Para calcular la probabilidad de una asignación concreta se multiplican todas las condiciones relevantes. Ejemplo:

$$P(+c,+i,-d) = P(+c)P(+i \mid +c)P(-d \mid +c)$$



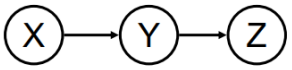
Una RB permite **reconstruir** cualquier entrada de la tabla de **probabilidades conjunta**, aunque no todas pueden representar todas las distribuciones conjuntas ya que la **topología** define qué condiciones de independencia se cumplen.

Una RB es una **codificación eficiente** de un **modelo probabilístico** de un dominio.

D-SEPARACIÓN

El objetivo es encontrar **(in)dependencias condicionales** en una RB. Para ello se debe analizar el grafo. Hay tres tipos de configuraciones de tripletes: cadena causal, causa común, efecto común.

La **cadena causal** tiene la siguiente estructura:
Si conocemos Y, X es independiente de Z.

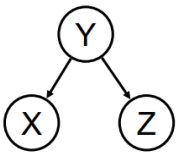


$$P(x,y,z) = P(x)P(y|x)P(z|y)$$

$$P(z|x,y) = \frac{P(x,y,z)}{P(x,y)} = \frac{P(x)P(y|x)P(z|y)}{P(x)P(y|x)} = P(z|y)$$

La evidencia en una cadena **bloquea** la influencia.

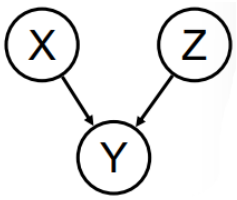
La **causa común** son dos efectos de la misma causa.
Dado Y, X y Z son independientes.



$$P(z|x,y) = \frac{P(x,y,z)}{P(x,y)} = \frac{P(y)P(x|y)P(z|y)}{P(y)P(x|y)} = P(z|y)$$

Observar la causa **bloquea** la influencia entre los efectos.

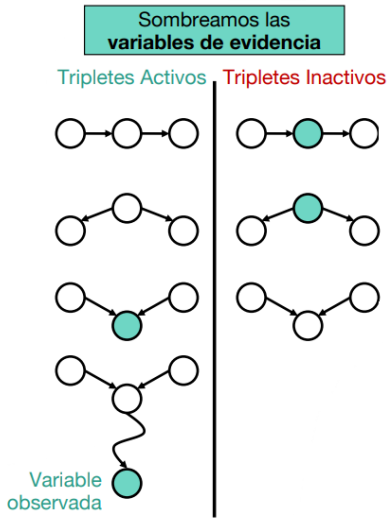
El **efecto común** son dos causas de un mismo efecto.



X y Z son independientes sí o sí, aunque dado Y no lo son.

Al revés que en los casos anteriores, observar un efecto **activa** la influencia entre las posibles causas.

Un **camino** es **activo** si todos sus tripletes son activos. Un solo triplete inactivo bloquea un camino.



8. RAZONAMIENTO PROBABILISTA

8.1. CONSTRUCCIÓN DE RB. CORRELACIÓN Y CAUSALIDAD

Las EB representan la **(in)dependencia condicional**. Pueden reflejar la **causalidad** real del dominio, aunque no necesitan ser **causales** ya que a veces no existe una red causal para el dominio, por lo que la red acaba teniendo flechas que reflejan **correlación**.

El algoritmo de construcción de una red bayesiana tiene los siguientes pasos:

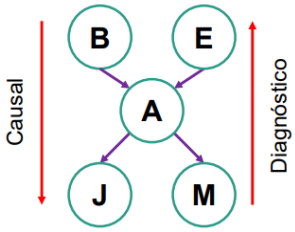
1. **Nodos**: determinar el conjunto de variables necesarias para modelar el problemas y **ordenarlas** $\{X_1,\dots,X_n\}$. La red será más compacta si las causas preceden a los efectos.
2. **Arcos**:

```
For i = 1 to n do:
  Elegir entre {X1, ... Xi-1} un conjunto mínimo de padres para Xi
  Para cada padre, insertar un arco del padre a Xi
  Escribir la tabla de probabilidades condicionales
P(Xi|padres(Xi))
End for
```

8.2. INFERENCIA PROBABILISTA

La **inferencia** se basa en responder a preguntas sobre probabilidad a partir de una RB. Para cada pregunta las variables se pueden dividir en 3 grupos:

- Q: nodos consulta (query)
- E: nodos evidencia
- H: nodos ocultos (hidden)



En dirección causal:
- E: nodos(s) raíz
- Q: nodo(s) hoja

En dirección diagnóstica:
- E: nodo(s) hoja
- Q: nodo(s) raíz

En la **inferencia por enumeración** la receta a seguir es:

1. Ver qué probabilidades incondicionales se necesitan para responder a la pregunta
2. Enumerar todas las probabilidades atómicas (para todos los posibles valores de las variables H)
3. Calcular suma de productos.

Este tipo de inferencia es muy lenta ya que se calcula la **distribución conjunta completa** antes de marginalizar a lo largo de las variables ocultas, por lo que es válida para redes pequeñas, pero para redes grandes es más fácil emplear la **inferencia por eliminación de variables**. La idea de la inferencia por EV es **entremezclar conjunción y marginalización**. Los pasos del algoritmo son:

1. **Ignorar** todas las variables que no sean un ancestro de una variable Q o E.
2. **Instanciar** las tablas de probabilidad con la evidencia.
3. **Eliminar** las variables ocultas H. Para ello:
   - Juntar todos los factores que mencionan a H.
   - Eliminar H (sumando)
4. **Juntar** todos los factores restantes y **normalizar**.

La **complejidad** de la eliminación de variables depende del **factor intermedio más grande que se genere**, que a su vez depende de:

- La **estructura** de la red:
  - Polytrees: si entre dos nodos cualesquiera hay como máximo un único camino:
    - La complejidad en tiempo y memoria es lineal con el tamaño de la red.
    - Si el nº de padres por nodo <k, es lineal con el nº de nodos.
  - Redes con conexiones múltiples: en el peor de los casos es exponencial
- El **orden** de eliminación de variables.

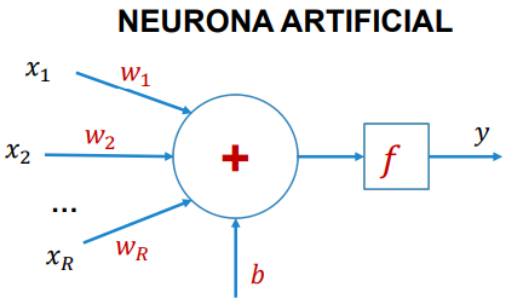
Finalmente, la **inferencia aproximada por muestreo** se basa en extraer N muestras de una distribución de muestreo S y calcular una distribución a posteriori aproximada. De esta forma la probabilidad estimada es **consistente**: converge a la verdadera probabilidad de P, cuando el número de muestras tienda a infinito. Puede servir en aprendizaje para obtener muestras de una distribución que no se conoce y en inferencia para generar muestras más rápido que calcular la respuesta exacta en redes complicadas.

9. PERCEPTRÓN MULTICAPA

9.1. MODELO DE NEURONA

Una **neurona artificial** consta de las siguientes partes:

- **Entradas de la neurona**: $x=(x_1,\ldots,x_R)^T$
- **Salida de la neurona**: y
- **Pesos**: $w=(w_1,\ldots,w_R)^T$
- **Offset/bias/sesgo**: b
- **Función de activación**: f

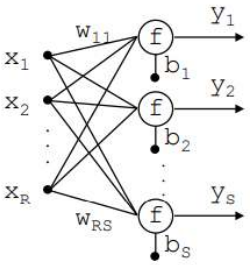


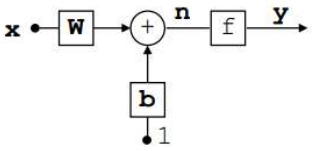
Una neurona con función de activación escalón f también se le llama perceptrón sigmoidal.

9.2. EL PERCEPTRÓN

Un **perceptrón** es un **clasificador lineal** ya que divide el espacio R-dimensional de entradas por un hiperplano y reconoce patrones sencillos.

Puede haber capas de más de una neurona:





$\mathbf{y} = f(\mathbf{W}\,\mathbf{x}+\mathbf{b})$

$$y_j = f\left(\sum_{i=1}^{R} w_{ij}x_i + b_j\right)$$

$$\mathbf{W} = \begin{pmatrix} w_{11}, w_{21}, \ldots w_{R1} \\ w_{12}, w_{22}, \ldots w_{R2} \\ \ldots \\ w_{1S}, w_{2S}, \ldots w_{RS} \end{pmatrix}$$

$$\mathbf{b} = (b_1, b_2, \ldots b_S)^T$$

$$\mathbf{x} = (x_1, x_2, \ldots x_R)^T$$

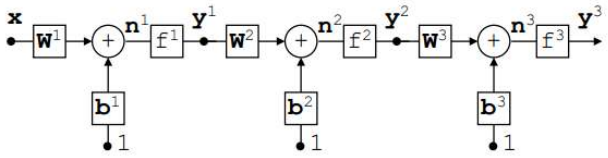
$$\mathbf{y} = (y_1, y_2, \ldots y_S)^T$$

Para realizar un **aprendizaje de los parámetros** (w y b) se utilizan los datos de entrenamiento (M muestras con salida deseada conocida (x,y_d)). El algoritmo es el siguiente:

```
Function perceptron_learn(x(1..M), y_d(1..M)) returns W, b
  W, b ← valores_aleatorios
  k ← 0
  repeat      {iterar con las M muestras de entrenamiento}
    k ← (k mod M) + 1
    y(k) ← f_esc(W x(k)+b)
    If y(k) <> y_d(k) then
      W ← W + r (y_d(k) - y(k)) x^T(k)      {r∈(0,1] factor de aprendizaje }
      b ← b + r (y_d(k) - y(k))
    end if
  until las M muestras bien clasificadas
  return W, b
```

9.3. REDES MULTICAPA

Cuando las salidas no son linealmente separables, se pueden crear redes con más capas (**capas ocultas**). La última capa debe tener el mismo número de neuronas que salidas.



$$\mathbf{y}^m = \mathbf{f}^m(\mathbf{W}^m\,\mathbf{y}^{m-1} + \mathbf{b}^m)$$

$$\mathbf{y}^3 = \mathbf{f}^3(\mathbf{W}^3\,\mathbf{f}^2(\mathbf{W}^2\,\mathbf{f}^1(\mathbf{W}^1\,\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

Ya no puede emplearse la regla del perceptrón para entrenar, así que el objetivo es **minimizar** una función de coste que mida nuestro error con los M datos de entrenamiento $\{x, y_d\}$.

$$\min_{\mathbf{W}_i, \mathbf{b}_i} \sum_{k=1}^{M} \mathcal{L}\left(f(\mathbf{x}^k), \mathbf{y}_d^k\right)$$

Se puede definir un **error cuadrático** de cada muestra:

$$\mathcal{L}(f(\mathbf{x}), \mathbf{y}_d) = \|f(\mathbf{x}) - \mathbf{y}_d\|^2 = \|\mathbf{y} - \mathbf{y}_d\|^2 = (\mathbf{y} - \mathbf{y}_d)^T(\mathbf{y} - \mathbf{y}_d)$$

Se modifican los pesos según el gradiente del coste:

$$\mathbf{W}_i(k+1) = \mathbf{W}_i(k) - r\frac{\partial \mathcal{L}}{\partial \mathbf{W}_i}$$

$$\mathbf{b}_i(k+1) = \mathbf{b}_i(k) - r\frac{\partial \mathcal{L}}{\partial \mathbf{b}_i}$$

$0 < r \leq 1$ Learning rate

Para calcular los gradientes se aplica la regla de la cadena de la derivación:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - r\;\mathbf{s}^m(\mathbf{y}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - r\;\mathbf{s}^m$$

10. REDES NEURONALES - ENTRENAMIENTO

El objetivo es **aprender una función** que predice el valor de las salidas, en función de las entradas: y=f(x). Los pasos a seguir son:

1. **Diseño** elegir la arquitectura:
   - Función de **coste**.
   - Función de **activación**.
   - Número de **capas**.
   - **Anchura** de cada capa.
2. **Entrenar** la red:
   - Elegir algoritmo de aprendizaje.
   - Controlar el sobreajuste.
3. **Evaluar** la red: validación cruzada.

- Hay miles de combinaciones
- No es posible comparar todas haciendo un grid-search
- Se hace búsqueda heurística guiada por la experiencia del diseñador
- Vamos a ver unas pocas reglas básicas

10.1. DISEÑO DE UNA RED NEURONAL

$$\hat{\theta} = \arg\min_{\theta} J(\theta)$$

FUNCIONES DE COSTE (LOSS, COST,...)

El entrenamiento busca los **pesos que minimizan el coste**.

Cross-Entropy:

- Obtiene solución de máxima verosimilitud.
- Equivale a minimizar la divergencia entre la distribución de los datos y la distribución aprendida.

$$J(\theta) = -\sum_{i \in data} log\ p(\mathbf{y}^{(i)}|\mathbf{x}^{(i)};\theta)$$

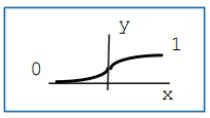
Depende de la función de activación de la capa de salida:

- Para capa lineal y error Gaussiano, J corresponde al **error cuadrático medio**.
- Para salidas sigmoidales o softmax, **cross-entropy** suele ir mejor.

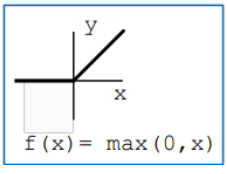
FUNCIONES DE ACTIVACIÓN

Hay varios tipos de funciones de activación de las **capas ocultas**:

- **Sigmoidal**: no va bien con redes profundas ya que los gradientes se desvanecen.

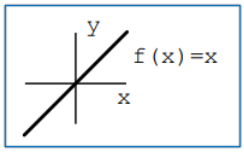


- **ReLu**: la más usada en la actualidad. Funciona muy bien en redes profundas. Es muy sencilla, y su derivada también. Tiene cierta base biológica ya que las neuronas reales tienen un umbral a partir del cual se activa.

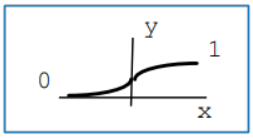


En cuanto a las funciones de activación de **salida**:

- Para **regresión**: lineal ya que no satura y va bien con el coste cuadrático.



- Para **clasificación binaria**: sigmoidal
  - Se puede interpretar la salida de la red como P(y=1|x)
  - Puede funcionar con coste cuadrático, pero va mejor con coste cross-entropy.



- Para **clasificación multi-clase**:
  - La neurona i representa P(y=i|x)
  - Las probabilidades están normalizadas (si usamos sigmoidales, la distribución no queda normalizada).
  - Va bien con coste cross-entropy.

$$softmax(\mathbf{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

TAMAÑO DE LA RED

Es el parámetro de diseño más importante. En general, a mayor profundidad y anchura puede funcionar mejor (aunque la red es más costosa de entrenar y más propensa a sobreajuste). Es mejor tener dos capas con menos neuronas que una con un nº de neuronas enorme.

10.2. ENTRENAMIENTO

RETROPROPAGACIÓN

```
r = rinicial
for m=1..M
  W^m y b^m = valores aleatorios
end for
repeat
  Emax = 0
  for cada muestra de entrenamiento
    for m = 1..M  {Salidas, hacia adelante}
      y^m = f^m(W^m y^(m-1)+b^m)
    end for
    e_y=(y_d-y)
    Emax:= max(Emax, e_y^T e_y)
    for m = M..1  {Sensibilidades, hacia atrás}
      if m = M then
        s^M = -2F^M' e_y
      else
        s^m = F^m' (W^(m+1))^T s^(m+1)
      end if
    end for
    for m = 1..M {Actualización de los pesos}
      W^m = W^m - r s^m (y^(m-1))^T
      b^m = b^m - r s^m
    end for
  end for {muestras}
until Emax < ε
```

Los requisitos para la retropropagación son:

- **Inicializar** los pesos con valores aleatorios (si inicializamos dos neuronas igual, siempre harán lo mismo).
- Las **derivadas de las funciones de activación** importan:
  - El escalón no sirve
  - Sigmoidal OK. Inicializar con pesos pequeños. En redes profundas los gradientes se desvanecen.
  - RELU MEJOR. Cuando está activada su gradiente es grande. La más utilizada para redes profundas.

OPTIMIZACIÓN

- **Descenso de gradiente estocástico**:
  - Descenso de gradiente con pequeños grupos de muestras.
  - Tamaño de minibatch de 16 (modelos grandes), 32-256
- Descenso de gradiente estocástico **con momentum**: se acuerda en qué dirección iba bajando.
- **Factor de aprendizaje adaptativo**.
- Métodos de 2º orden:
  - Calculan o aproximan el Hessiano (segunda derivada del coste).
  - Son muy pesados para redes grandes.

FACTOR DE APRENDIZAJE ÓPTIMO

Aproximando la función de coste por una cuadrática puede acelerarse la convergencia estimando la segunda derivada y adaptando el factor de aprendizaje para cada peso.

EVITAR EL SOBREAJUSTE

Para evitar el sobreajuste se puede:

- **Regular** J (añadiendo a J una **penalización** por los pesos).
- Conseguir más datos (**data augmentation**).
- Compartir parámetros.
- **Dropout** (apagar aleatoriamente una fracción de las neuronas).
- ***Early stopping***: iterar demasiado provoca sobreajuste

11. REDES NEURONALES - EVALUACIÓN

Los tipos de errores y métricas de la evaluación se pueden ver en el Apartado 6.3.

La **generalización** es la capacidad de predecir la salida para nuevos datos.

K-FOLD CROSS-VALIDATION

Se basa en separar los datos de entrenamiento en k partes de forma que en cada una de las k iteraciones, cada una de esas partes funciona como datos de validación y el resto de partes como entrenamiento.

```
Function kfold_cross_validation(Learner, k, examples) returns hypothesis
  best_size ← 0;  best_errV ← inf;
  for size = 1 to n do          {para los distintos valores de los hyper-parámetros}
    err_T ← 0;  err_V← 0
    for fold = 1 to k do                  {separar N/k ejemplos para validación}
      [training_set, validation_set] ← Partition(examples, fold, k)
      h ← Learner(size, training_set)                  {aprender con el resto}
      err_T ← err_T + Error(h, training_set)
      err_V ← err_V + Error(h, validation_set)
    end for
    err_T ← err_T/k;  err_V← err_V/k       {calcular el error medio de las k veces}
    if has_converged(err_T) and err_V < best_errV then
      best_size ← size          {guardar el mejor valor de los hyper-parámetros}
    end if
  end for
  return Learner(best_size, examples)              {aprender de nuevo con todos}
```

12. DEEP LEARNING

12.1. CONVOLUCIÓN

La **convolución** es aplicar un Kernel de tamaño nxn (matriz de pesos fija) a cada grupo de nxn píxeles vecinos de una imagen:

- Cada píxel de salida depende de nxn píxeles de entrada.
- Es una operación local.

Las **redes neuronales convolucionales** tienen **ReLu, data augmentation y dropout**.

12.2. ¿QUÉ SE PUEDE HACER CON EL DEEP LEARNING?

- Clasificación de imágenes.
- Detección de caras.
- Reconocimiento facial.
- Aprender estrategias de juego a base de jugar.
- Conducción autónoma.
- Sistemas de recomendación.
- Procesamiento de lenguaje.

Con LiiNCE, lo único que te va a costar es estudiar

-80% de descuento en tus gafas graduadas

Gafas graduadas desde 15€

Pide cita online en **liince.com**

Esquina de San Ignacio de Loyola con Lacarra de Miguel

LiiNCE