1. Búsqueda

1.1. Búsqueda ciega e informada

- Problema: encontrar la ruta con coste mínimo desde uno nodo inicial a un nodo final.
  - El grafo que representa el problema tiene inconvenientes: puede ser infinito, puede haber varios nodos finales, ...
  - Enfocar la búsqueda en un árbol cuya raíz es el estado inicial.
    - Cada nodo del árbol es un *estado* que se corresponde con un nodo del grafo. La manera de llegar a un estado los diferencia con respecto a los nodos del árbol.
    - **Test objetivo** para saber si un estado es final o meta.
    - **Acción o expansión de un nodo** para obtener los sucesores de un estado del árbol
    - **Lista de abiertos** con los estados descubiertos pero no explorados.
    - **Estrategia** define la ordenación de los nodos en la lista de abiertos.
    - **Utilidad** g da el coste desde el nodo raíz hasta el actual (pero considerando los estados del árbol, no solo los nodos del grafo).
- Una **heurística** h es una función $h : V \to [0, +\infty)$ donde V son los nodos. h estima la distancia a la meta y se normalmente se obtiene por relajación del problema.
  - h se dice **monótona** $\iff \forall n, n',\ h(n) \leq h(n') + \Gamma(n, n')$ (desigualdad triangular)
  - h se dice **admisible** $\iff \forall n,\ h(n) \leq h^*(n)$ donde $h^*(n)$ es el **coste real óptimo** de n a la meta.
    - h monótona $\implies$ h admisible
    - Conocer h^* normalmente requiere resolver el problema, por eso es más fácil demostrar h monótona que h admisible.
- Definimos $f = g + h$ para cada nodo.

Búsqueda genérica en árbol

```
function búsqueda-en-árbol (problema, estrategia)
;; devuelve solución o fallo
;; lista-abierta contiene los nodos de la frontera del árbol de búsqueda
Inicializar árbol-de-búsqueda con nodo-raíz
Inicializar lista-abierta con nodo-raíz
Iterar
  If (lista-abierta está vacía) then return fallo
  Elegir de lista-abierta, de acuerdo a la estrategia, un nodo a expandir.
  If (nodo satisface test-objetivo)
    then return solución (camino desde el nodo-raíz hasta el nodo actual)
    else eliminar nodo de lista-abierta
      expandir nodo
      añadir nodos hijo a lista-abierta
```

Búsqueda en grafo o *con eliminación de estados repetidos*: añadir una lista de cerrados que contiene los nodos ya explorados (= expandidos). No se introducen en la lista de abiertos aquellos nodos que ya estén en la lista de cerrados. La **estrategia** determina la ordenación de la lista abierta:

- FIFO (cola): **búsqueda en anchura**.
- LIFO (pila): **búsqueda en profundidad**.
- Por valor de g ascendente: **Dijkstra o coste uniforme**
- Por valor de h ascendente: **búsqueda avariciosa**
- Por valor de f ascendente: A^*

¿Qué queremos?

- **Completitud**: encontrar la solución siempre que exista.
- **Optimalidad**: encontrar siempre la solución de menor coste g.
- A^* (A-estrella): ordenar la lista de abiertos por valor de $f = g + h$ ascendente.
  - A^* sin eliminación de estados repetidos (= búsqueda en árbol) y h admisible es completa y óptima.
  - A^* con eliminación de estados repetidos (= búsqueda en grafo) y h monótona es completa y óptima.

1.1.1. Coste computacional

- b **factor de ramificación**: el mayor número de sucesores de un estado (suponemos $b < \infty$)
- m profundidad máxima del árbol de búsqueda (suponemos $m < \infty$)
- d profundidad del nodo objetivo más superficial
- C^* coste del camino de la solución óptima
- $\varepsilon \geq 0$ coste mínimo de un acción

	Tiempo	Memoria	Completa?	Óptima?
en anchura	$O(b^d)$	$O(b^d)$	Sí[1]	Sí[2]
en profundidad	$O(b^m)$	$b \cdot m + 1$	No	No
Dijkstra	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	Sí ($\varepsilon > 0$)	Sí
avariciosa	$O(b^m)$	$O(b^m)$	No	No
A^*	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	Sí*	Sí*

1.2. Búsqueda entre adversarios

1.2.1. Clasificación de problemas de búsqueda (= juegos)

- Número de jugadores: solo nos interesan aquellos con dos jugadores.
- **Suma cero, suma constante o suma variable**. *Suma* se refiere a sumar los valores de la utilidad desde el punto de vista de min o de max.
  - Asignar los valores `perder = -1, empatar = 0, ganar = 1` en el ajedrez da un juego de suma cero ya que si uno pierde, el otro gana y por tanto sus valores de utilidad suman 0. Ocurre lo mismo si los dos empatan.
  - Asignar los valores `perder = 0, empatar = 1, ganar = 2` en el ajedrez da una juego de suma constante ya que si una pierde y el otro gana la suma de las utilidades desde ambos puntos de vista es 2. Ocurre lo mismo si los dos empatan ($1 + 1 = 2$).
  - Los juegos de suma variable no son susceptibles de ser atacados por búsqueda entre adversarios.
- **Información perfecta** (ajedrez, damas) o **información parcial** (casi todos los juegos de cartas).
- **Deterministas** (ajedrez, damas) o **estocásticos** (backgammon).
- Tiempo y número de movimientos (limitados o ilimitados).

1.2.2. Minimax

- Modelización de un problema con dos jugadores.
- Al que juega primero le llamamos *max*, al otro *min*.
- Esta estrategia encuentra la jugada óptima para max.
- Definimos el valor `minimax(n)` para un nodo:

$$\text{minimax}(n) \equiv \begin{cases} \text{Utilidad}(n) & \text{si } n \text{ terminal} \\ \text{máx}\{\text{minimax}(s) : s \text{ sucesor de } n\} & \text{si } n \text{ es un nodo max} \\ \text{mín}\{\text{minimax}(s) : s \text{ sucesor de } n\} & \text{si } n \text{ es un nodo min} \end{cases}$$

- Optimalidad: minimax es óptimo si el oponente lo es. Si no lo es hay maneras mejores de ganarle (esto es peligroso).
- Complejidad temporal $O(b^m)$ y espacial $O(b \cdot m)$.
- Con **poda** $\alpha - \beta$:
  - En nodos `min` se actualiza $\beta = \text{mín}(\beta, \alpha_i \text{ de los hijos })$
  - En nodos `max` se actualiza $\alpha = \text{máx}(\alpha, \beta_i \text{ de los hijos})$
  - Es útil hacer minimax sin poda para los ejercicios ya que permite comprobar si los intervalos $[\alpha,\ \beta]$ están bien: en nodos max, el valor minimax coincide con α y en nodos min, el valor minimax coincide con β.
  - Complejidad temporal: depende de la ordenación de la búsqueda. Es mejor si los nodos mejores se exploran primero.
    - En el caso peor no hay mejora.
    - En el caso medio (ordenación aleatoria: $O(b^{3d/4})$
    - En el caso mejor (ordenación perfecta: $O(b^{d/2})$

2. Lógica de predicados

2.1. Formalización:

Hay que acordarse de:

- Hay **constantes, variables, predicados y funciones**.

[1] Búsqueda en anchura solo es completa y óptima si el coste es una función no decreciente de la profundidad.
[2] Búsqueda en anchura solo es completa y óptima si el coste es una función no decreciente de la profundidad.

- Un **predicado** devuelve un valor de verdad mientras que una **función** devuelve otro átomo. Por ejemplo: `mejorAmigoDe(perso` es una función mientras que `ViveEn(ciudad, persona)` es un predicado.
- Nunca* se pone un $\forall$ con un $\wedge$ y tampoco se pone un $\exists$ con un $\implies$.
- Nunca se pone un predicado dentro de otro o de una función.
- Las definiciones utilizan un $\iff$.
- Si tenemos dos opciones normalmente hay que especificar que son distintas.

2.2. Ejercicios

2.2.1. Hoja 2, 2018: ejercicio 2

1. Dos nodos son hermanos si, siendo distintos, tienen el mismo padre.

$$\forall x, y[(\neg I(x, y) \wedge I(padreDe(x), padreDe(y))) \iff H(x, y)]$$

2. Un camino entre dos nodos es una secuencia de uno o varios enlaces entre dichos nodos.

$$\forall x, y, c[C(c, x, y) \iff (I(c, enlace(x, y)) \vee \exists z, m, n(\neg I(m, n) \wedge C(m, x, z) \wedge C(n, z, y)]$$

2.2.2. Parcial 1, 2014-2015: ejercicio 3

1. Ejemplo
2. Se puede diseñar una máquina de Turing para computar la solución de cualquier problema que pueda ser resuelto mediante la aplicación de un algoritmo sobre unos datos de entrada.
3. Una máquina de Turing universal puede simular la acción de cualquier máquina de Turing sobre los datos almacenados en su cinta

$$\forall u[Universal(u) \implies \forall t, d(comp(t, d) = comp(u, descr(t_2, d))]$$

2.3. Incertidumbre

Problema: dado un conjunto de pares (atributos, clase) donde atributos es un vector, elaborar un modelo que permita asignar una clase de entre un conjunto de clases a otros vectores de atributos.
Definiciones:

- El prior $P(\text{clase})$
- La evidencia $P(\text{atributos})$
- La verosimilitud $P(\text{atributos} \mid \text{clase})$
- El posterior $P(\text{clase} \mid \text{atributos}$. Para un vector de atributos dado, la suma de los posteriores sobre cada clase da siempre 1, es decir, $\sum P(\text{clase}_i \mid \text{atributos}) = 1$.

Modelos:
Un modelo de predicción nos asigna una clase a un vector de atributos dado en base a los datos (pares (atributos, clase)) de los que partimos.

- Modelo basado en priores: predecir, para cualquier vector de atributos, la clase con mayor prior (ignorar los atributos de un dato para clasificarlo).
- **ML = Maximum Likelihood** o Máxima verosimilitud: predecir la clase que maximiza $P(\text{atributos} \mid \text{clase})$.
- **Calsificador de Bayes o MAP** o maximizar los posteriores: predecir la clase que maximiza $P(\text{clase} \mid \text{atributos})$. *Bayes es óptimo = minimiza el error.*
- **Clasificador Naïve Bayes:** asume que los atributos son independientes entre sí y por tanto solo dependen de la clase, difiere del clasificador de Bayes en la manera de descomponer $P(\text{clase} \mid \text{atributos})$:

$$P(\text{clase} \mid \text{atributos}) = \frac{P(\text{atributos} \mid \text{clase}) \cdot P(\text{clase})}{P(\text{atributos})} = \frac{(\prod P(\text{attributo}_i \mid \text{clase}))P(\text{clase})}{P(\text{atributos})}$$

  Predice la clase que maximiza $P(\text{clase} \mid \text{atributos})$ como hace MAP.

- **Clasificador según una red bayesiana:** dadas las dependencias entre los atributos(el grafo) descomponemos $P(\text{clase} \mid \text{atributos})$ según estas. Predice la clase que maximiza $P(\text{clase} \mid \text{atributos})$ como hace MAP.

Nota: si tenemos clases uniformes, es decir, si $P(\text{clase}_i) = P(\text{clase}_j)$ para toda clase de nuestro problema, entonces MAP y ML predicen siempre la misma clase (no necesariamente con la misma probabilidad).

2.4. Aprendizaje automático

Construir de manera automática modelos para clasificación.
Dado un conjunto de entrenamiento formado por pares (atributos, clase) donde atributos es un vector, generar un modelo que permita asignar una clase de entre un conjunto de clase a otros vectores de atributos.

2.4.1. Árboles de decisión

- Problema: minimizar la profundidad del árbol.
- Solución: poner más arriba decisiones que aporten más información, es decir, poner como raíz del árbol la decisión sobre el atributo que maximice la ganancia de información.

Dos algoritmos muy parecidos:

- ID3: genera árboles propiamente dichos a partir de datos que pueden ser cualitativos. Un nodo del árbol representa la decisión basada en un atributo y por tanto tiene tantas ramas como valores toma el atributo. Permite atributos cualitativos.
- C4.5: genera particiones del espacio R^n donde n es el número de atributos. En cada iteración nos da una partición del dominio del atributo tal que se maximiza la ganacia de información. Requiere atributos cuantitativos (que pueden obtenerse asignando valores a los atributos cualitativos pero entonces no se diferencia de ID3).

2.4.2. k-NN = k vecinos más próximos

- Consiste en codificar los atributos numéricamente de manera que sean puntos en R^n. Para clasificar un punto dado, se selecionan los k puntos más próximos del conjunto de entrenamiento según una métrica. La clasificación es la moda de las clases de los puntos cercanos, por eso es conveniente evitar empates, por ejemplo, tomando $k = 3$ si hay dos clases.