# Trabajo teoría de juegos

Manuel García
28-11-2022

# Contenido

# 1. ¿Qué es la IA?

La inteligencia artificial o IA es, dentro de la informática, un campo de estudio encargado de desarrollar programas o algoritmos capaces de replicar operaciones atribuidas a la inteligencia humana.

# 2. Historia de la IA

En 1950 Alan Turing asienta las bases de la informática actual, consolida el campo de la inteligencia artificial y crea el **test de Turing**, que determina si una maquina es considerada inteligente o no.

En 1956, en la conferencia de Darmouth, se acuña el termino **Inteligencia Artificial**.

En 1957, Frank Rosenblat diseña la primera red neuronal artificial.

En 1966, el MIT desarrolla uno de los primeros programas (ELIZA) en procesar lenguaje natural y conversar a través de una serie de frases programadas.

EN 1979, un ordenador gana al campeón mundial de backgammon, lo que impulsa nuevos proyectos para juegos más complejos.

En 1981 Japón comienza el proyecto “Quinta Generación”, en el cual, este país invierte en una nueva generación de ordenadores que utilicen inteligencia artificial.

En 1987, **Martin Fischles** y **Oscar Firschein** describen atributos de un agente inteligente, lo cual abre nuevas áreas de investigación para la IA.

En 1996, la supercomputadora **Deep Blue** vence a Kasparov en una partida de ajedrez.

En 2005, usando la **Ley de Moore**, se predijo que las maquinas alcanzaran un nivel de inteligencia humano en 2029

En 2012, **Google** crea un superordenador capaz de aprender a través de YouTube a identificar caras y cuerpos humanos.

En 2014, un ordenador supera con éxito el **Test de Turing**.

En 2016, **Microsoft** lanza **Tal**, un chatbot capaz de aprender a partir de la interacción con las personas. Ese mismo año un ordenador de **Google** venció al campeón del juego “Go”.

En 2017, un software se impone ante rivales humanos en un torneo de póker.

## 3. ¿Qué es la Teoría de Juegos?

La **Teoría de Juegos** es el estudio matemático en el cual se toman decisiones teniendo en cuenta las elecciones que hacen otros. Esta teoría ha contribuido en la comprensión de la conducta humana frente a la toma de decisiones.

## 4. Completitud

Propiedad que garantiza o no encontrar la solución si es que existe.

## 5. Complejidad temporal

En informática, es la propiedad computacional que define la eficiencia de un algoritmo, es decir, el tiempo que tarda y el número de operaciones optimas que realiza un algoritmo.

## 6. Complejidad espacial

Se trata de la cantidad de memoria que necesita un algoritmo para llegar a completarse.

## 7. Optimidad

Mejor manera de realizar algo para llegar a resultado correcto.

# 8. Tipos de búsquedas

## Búsqueda tentativa

Se avanza en una dirección, y si se llega a un punto que no llega a alguna meta, se abandona el camino y se retoma en alguno anterior que también prometía.

## Búsqueda irrevocable

Una vez tomado un camino, no se puede abandonar

## Búsqueda ciega

Este tipo de búsqueda es un método de tentativa, que intenta encontrar la primera solución sin importar lo optima que sea, utilizando solo la información acerca de si un estado es o no objetivo como guía en el proceso de búsqueda.

Tipos de búsqueda ciega:

Búsqueda en anchura:

- Siempre encontrará la solución y además será óptima.
- Consume más recursos por la alta complejidad espacial y temporal.

Búsqueda en profundidad:

- Menor complejidad espacial
- No siempre encontrara la solución o encontrara una solución muy alejada si esta sin acotar
- Posibilidad de que aparezcan bucles infinitos

Búsqueda de costo uniforme:

- Recorre el camino entre la raíz y el destino que tenga coste mínimo.
- Empieza por la raíz y va visitando el nodo con menor costo y así hasta el final, por lo cual encontrara la solución que sea óptima.

Búsqueda profundidad limitada

- Se define una profundidad predefinida
- Se desarrolla el árbol realizando una búsqueda en profundidad hasta el límite definido anteriormente y si se encuentra la solución finaliza, en caso contrario se establece un nuevo límite y se comienza otra vez

Búsqueda bidireccional

- Se llevan a cabo dos búsquedas a la vez: una descendente desde el inicio y otra ascendente desde el nodo meta
- Al menos una de estas búsquedas debe ser en anchura para que los dos recorridos puedan encontrarse en algún momento
- En el momento en que se llegue a un nodo ya visitado con el otro tipo de búsqueda, el algoritmo acaba
- El camino a la solución es la suma de los caminos hallados por cada búsqueda desde un nodo hasta el inicial y hasta el meta

## Búsqueda heurística

Busca soluciones aceptables mediante la reducción del espacio de búsqueda y es capaz de determinar la proximidad de la solución y la optimidad de la misma utilizando la información sobre lo prometedor que es un nodo para llegar a él desde la solución, para ello se utilizara la función heurística (F(n)=g(n)+h(n)).

Tipos de búsqueda heurística:

Búsqueda tacaña:

- Elige como siguiente nodo aquel con mayor función de evaluación
- Se trata de una búsqueda de tipo tentativo que no depende en exceso de la función de evaluación, pero, tiene cuan complejidad espacial excesiva, ya que, se deben guardar todos los nodos abiertos

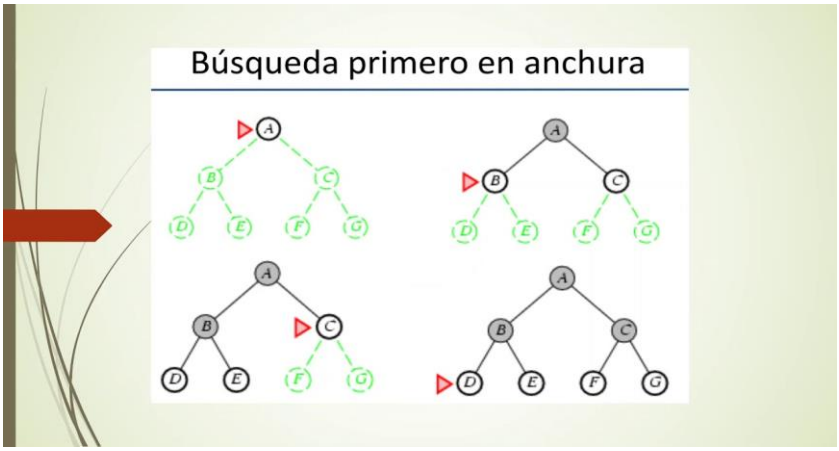
Búsqueda A*:

- Siempre encontrara la solución que sea más óptima y de menor coste
- Complejidad espacial elevada.

## 9. Búsqueda en anchura

Procede a buscar el recorrido nivel a nivel, es decir, no expande ningún nodo antes de haber expandido todos los del nivel anterior. Se implementa con una estructura FIFO (First in, first out).

Este tipo de búsqueda asegura que, de haber solución, será una solución óptima.



Su pseudocodigo sería el siguiente:

```
ABIERTA = [nodoInicial]
while noEsVacia(ABIERTA) do
        actual = primero(ABIERTA)
        if esObjetivo(actual) then
                return camino(nodoInicial, actual)
        succ = sucesores(actual)
        foreach s de succ do
                s.padre = actual
                s.coste = actual.coste + coste(actual,s)
                añadir(s, ABIERTA, final)
        end do
end do
return "No encontrado"
```

## 10. Búsqueda en profundidad

La búsqueda se realiza por una sola rama del árbol hasta encontrar una solución o hasta que se tome la decisión de terminar la búsqueda por esa dirección. Si esto ocurre se produce una vuelta atrás y se sigue por otra rama hasta visitar todas las ramas del árbol si es necesario a menos que le indique un rango de nodos máximos a visitar.


Este tipo de búsqueda es de menor complejidad que la de anchura, pero la solución podría no ser encontrada o que este muy alejada en cuanto a recorrido.

La representación en pseudocódigo sería la siguiente:

```
ABIERTA = [nodoInicial]
while noEsVacia(ABIERTA) do
        actual = primero(ABIERTA)
        if esObjetivo(actual) then
                return camino(nodoInicial, actual)
        if actual.profundidad < limite then
                succ = sucesores(actual)
        else succ = <vacio>
        borrar(actual, ABIERTA)
        foreach s de succ do
                s.padre = actual
                s.coste = actual.coste + coste(actual,s)
                s.profundidad = actual.profundidad + 1
                añadir(s, ABIERTA, principio)
        end do
end do
return "No encontrado"
```

## 11. Búsqueda A*

Se trata de un método de búsqueda de tipo tentativo, que pondera a la vez lo cerca que estamos del nodo meta y lo lejos que estamos del nodo inicial.

Este tipo de búsqueda encontrara siempre el resultado que tenga el camino con el menor coste entre inicio y meta.

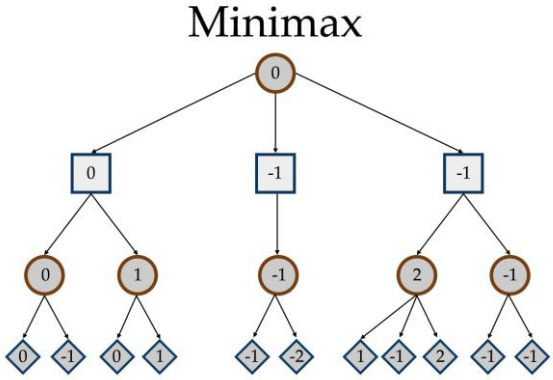
| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | 19 | 20 | 21 | 22 |
| 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | 18 | 19 | 20 | 21 |
| 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | 17 | 18 | 19 | 20 |
| 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 16 | 17 | 18 | 19 |
| 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | 15 | 16 | 17 | 18 |
| 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | 14 | 15 | 16 | 17 |
| 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | 13 | 14 | 15 | 16 |
| 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 12 | 13 | 14 | 15 |
| 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Su pseudocodigo sería el siguiente:

```
1:    ABIERTA = [nodoInicial]
2:    while noEsVacia(ABIERTA) do
3:        actual = primero(ABIERTA)
4:        if esObjetivo(actual) then
5:            return camino(nodoInicial, actual)
6:        succ = sucesores(actual)
7:        foreach s de succ do
8:            if esta(s, ABIERTA) then
9:                comprobarCamino(s, actual)
10:           else
11:               s.padre = actual
12:               s.g = actual.g + coste(actual,s)
13:               s.h = <obtener valor heurístico de s>
14:               introducir Ordenado(s, ABIERTA)
15:       end do
16:   end do
17:   return "No encontrado"
```

## 12. Explicación de juegos minimax y poda alfa-beta.

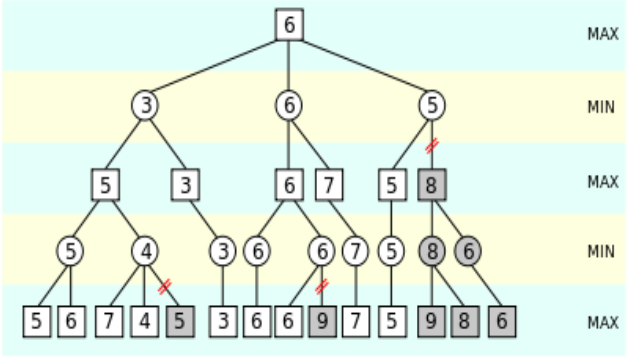
Un algoritmo minimax es un método de decisión para minimizar la perdida máxima en un juego con un adversario y con toda la información. Ejemplos donde se podría aplicar este algoritmo serian juegos como el ajedrez, Go o las damas, juegos de dos jugadores, por turnos, donde ambos jugadores saben que posibles movimientos se podrían hacer, suponiendo siempre que el contrincante elegirá el peor movimiento para ti y viceversa.




El pseudocodigo sería el siguiente:

```
MINIMAX(posicion, nivel)
/* casos base */
if (esGanador(posicion)) then
        devolver +∞
else if (esPerdedor(posicion)) then
        devolver −∞
else if (esEmpate(posicion)) then
        devolver 0
else if (nivel = limite) then
        devolver evaluacion(posicion)
else
        /* caso recursivo */
        for all sucesor i de posicion do
                valores[i] := MINIMAX(sucesor i, nivel+1)
                if (esNodoMAX(nivel)) then
                        devolver maximo(valores)
                end if
                if (esNodoMIN(nivel)) then
                        devolver minimo(valores)
                end if
        end for
end if
```

Para reducir el coste computacional de minimax, se usa un método de poda alfa-beta que nos lleva a una solución peor a las ya encontradas. Con este método decidimos que ramas no serán exploradas, ahorrando espacio y tiempo computacional.




Los valores alfa son los calculados al ir hacia atrás en los nodos max. Estos valores no pueden decrecer.

Los valores beta son, al contrario, son los calculados hacia atrás en los nodos min y no pueden crecer.

El pseudocodigo de la poda alfa-beta sería el siguiente:

```
función alfa-beta(nodo //en nuestro caso el tablero, profundidad, α, β, jugador)
    si nodo es un nodo terminal o profundidad = 0
        devolver el valor heurístico del nodo
    si jugador1
        para cada hijo de nodo
            α := max(α, alfa-beta(hijo, profundidad-1, α, β, jugador2))
            si β≤α
                romper (* poda β *)
        devolver α
    si no
        para cada hijo de nodo
            β := min(β, alfa-beta(hijo, profundidad-1, α, β, jugador1))
            si β≤α
                romper (* poda α *)
        devolver β
```