

## Práctica 1: Algoritmo de búsqueda elemental



Víctor López Moreno Inteligencia artificial



## 1. A\* Search

Este método implementa el algoritmo de búsqueda A\* para encontrar el camino más corto hacia el jaque mate en un juego de ajedrez. Comienza inicializando el estado actual y la frontera, que es una lista de los nodos que están por explorar. El estado actual se añade a la lista de estados visitados y a la frontera con su valor de función de evaluación, que es la suma de su costo de camino (g) y su valor heurístico (h).

El algoritmo entra en un bucle mientras la frontera no esté vacía, es decir, mientras haya nodos por explorar. En cada iteración, selecciona el nodo con el valor de función de evaluación más bajo y lo elimina de la frontera. Si este nodo representa un estado de jaque mate, el algoritmo termina, imprime la profundidad mínima para alcanzar el estado objetivo, y reconstruye el camino hacia el estado inicial.

```
def AStarSearch(self, currentState):

# Utilizamos el dictPath para obtener el g(n) de los nodos (g(n)=depth)
self.dictPath[str(currentState)] - (None, -1)

# Inicializamos la Frontera
frontera - []
self.listVisitedStates.append(currentState) # Añadimos en alcanzados currentState
gCurrentState - 0 # Inicializamos el coste g(n) del currentState
# Añadimos en la frontera el currentState y se (n) - g(n) + h(n)
frontera.append((self.h(currentState), currentState))

# Mientras la frontera no este vacía
while len(frontera) > 0:
# Expandimos nodo con f(n) mínimo de la frontera
fn, nodo = min(frontera, key=lambda x: x[9])
frontera.remove(fin, nodo)) # Elisimamos este nodo

# Sacamos el valor g(n) de los sucesores del nodo
gson - self.dictPath[str(nodo)][1] + 1
# Si g(n) > 0 -> no es el estado inicial (root)
if gson > 0:
# Movemos las piezas desde el currentState al nodo
self.movePieces(currentState, gCurrentState, nodo, gson)

# Comprovamos si al nodo realiza jaque mate
if self.isChecNtex(endo):
# Imprissimos la profundidad para obtener el estado objetivo
print("Profundida dinfima para obtener el estado objetivo
print("Profundida finima para obtener el estado objetivo
# ReconstructPath(nodo, gson)
break

# Expandimos los hijos/sucesores del nodo minimo
for son in self.getListNextStatesN(nodo):
# Obtenemos hin) del hijo/sucesores
haon - self.hi(son)
# Si el hijo no esta en alcanzado o si su camino es mas eficiente
if not self.isVisited(son) or (hson + gson < fn):
# Guardamos el alcanzado el hijo
self.listVisitedStates.append((floson + gson)
# Añadimos el hijo a la frontera con su f(n)
frontera.append((floson + gson), son))

# Guardamos el nodo, en currentState para la próxima iteración
currentState - nodo
@ Guardamos (o) del nodo, en gCurrentState para la próxima iteración
gCurrentState - gson
```

Si el nodo no es el estado de jaque mate, el algoritmo genera sus estados sucesores y los evalúa. Para cada sucesor, calcula su valor heurístico y comprueba si ya ha sido visitado o si su camino es más eficiente que el actual. Si es así, añade el sucesor a la lista de estados visitados y a la frontera con su nuevo valor de función de evaluación. También guarda el nodo sucesor y su costo de camino para la próxima iteración.

Finalmente, el algoritmo actualiza el estado actual y su costo de camino para la próxima iteración. Este proceso se repite hasta que se encuentra un estado de jaque mate o hasta que se han explorado todos los posibles estados.



## 2. ¿Sus algoritmos funcionan exactamente de la misma manera? ¿ Tuvo que cambiar algo ?

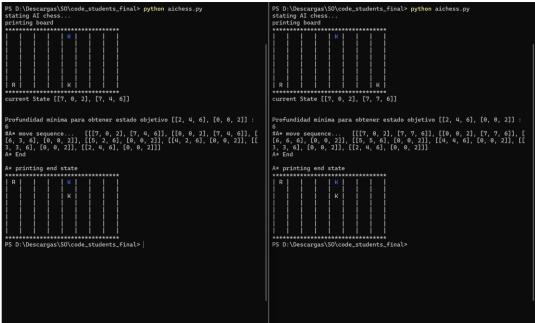


Figura 1. Ejecución código aichess.py

Ejecutamos el código, sin ninguna modificación con dos variaciones de posiciones, en el rey blanco. En la figura de la izquierda nos encontramos con la ejecución del código con el rey blanco con la posición inicial [7,4] del tablero. En cambio en la figura de la derecha nos encontramos que el rey blanco tiene como posición inicial el [7,7].

Como podemos observar la profundidad mínima para alcanzar el jaque mate es de 6 movimientos tanto en las posiciones iniciales [7,4] como en [7,7]. Sin embargo, la secuencia de movimientos varía según la disposición de las piezas en el tablero.

El número de nodos recorridos muestra una notable diferencia: en la configuración [7,4], se recorren 377 nodos, mientras que en [7,7] solo 277 nodos. Esto indica que el espacio de búsqueda es más amplio en la primera configuración, posiblemente debido a una mayor cantidad de movimientos y estados sucesores que deben evaluarse.

En conclusión, aunque el algoritmo A\* siempre encuentra un camino óptimo con la misma profundidad mínima, el número de nodos explorados puede variar significativamente, dependiendo de la configuración de las posiciones iniciales.