

## Práctica 1: Algoritmo de búsqueda elemental



## 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 su f(n) = g(n) + h(n)  
    frontera.append((self.h(currentState), currentState))  
  
    # Mientras la frontera no este vacia  
    while len(frontera) > 0:  
        # Expandimos nodo con f(n) minimo de la frontera  
        fn, nodo = min(frontera, key=lambda x: x[0])  
        frontera.remove((fn, nodo)) # Eliminamos 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.movePiezas(currentState, gCurrentState, nodo, gson)  
  
        # Comprobamos si el nodo realiza jaque mate  
        if self.isCheckMate(nodo):  
            # Imprimimos la profundidad para obtener el estado objetivo  
            print("Profundidad mínima para obtener estado objetivo", str(nodo), ":", gson)  
            # Reconstruimos el camino  
            self.reconstructPath(nodo, gson)  
            break  
  
        # Expandimos los hijos/sucesores del nodo minimo  
        for son in self.getListNextStatesM(nodo):  
            # Obtenemos h(n) del hijo/sucesores  
            hson = self.h(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(son)  
  
                # Guardamos su g(n)  
                self.dictPath[str(son)] = (nodo, gson)  
  
                # Añadimos el hijo a la frontera con su f(n)  
                frontera.append((hson + gson, son))  
  
    # Guardamos el nodo, en currentState para la próxima iteración  
    currentState = nodo  
    # Guardamos g(n) 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.

