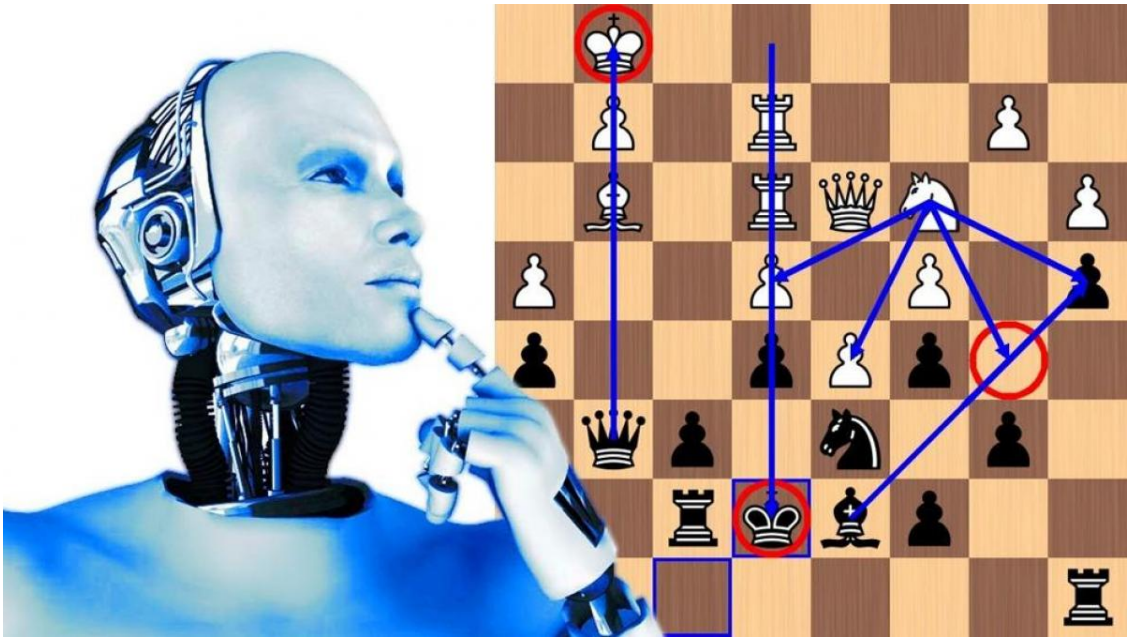
Universitat de Barcelona

Facultat de Matemàtiques i Informàtica

Pràctica 1: Elementary Search Algorithms

Intel·ligència Artificial



Carlos Bellanco Ortiz i Víctor Sort Rubio

Professors: Dr. Ignasi Cos Aguilera i Dra. Maite López-Sánchez

Contents

1 Introducció

Els objectius principals d'aquesta pràctica consisteixen en familiaritzar-nos amb els moviments bàsics del joc d'escacs, així com amb algorisme bàsics d'intel·ligència artificial. En particular, donat un taulell d'escacs prou simple, amb només un rei blanc, una torre blanca i un rei negre que no es pot moure, se'ns demana implementar un algorisme d'A* per aconseguir un escac i mat. L'heurística que hem d'usar ja ens ve donada inicialment. Després, però, se'ns demana si hem de modificar el codi en el cas de tenir una altra configuració de taulell.

2 Explicacions i respostes

2.1 Heurística utilitzada

Tot i que la heurística que hem usat per implementar l'algorisme A* ja ens venia donada, cal comprovar igualment que es tracti d'una heurśtica òptima pel problema.

Donat que només es tenen 3 peces al joc i el rei negre està fix a la posició [0,4], perquè hi hagi un escac i mat s'ha de tenir la torre blanca a la primera fila per amenaçar el rei, evitant les columnes contigües, doncs el rei negre la podria matar, i el rei blanc a la posició [2,4] perquè el rei negre no es pugui escapar en cap direcció.

Tenint en compte l'anterior, l'explicació de l'heurística és senzilla. Simplement es retorna en quants moviments poden estar les peces en aquesta posició. En el cas de la torre, pot ser 0 (si ja hi és), 1 (si només s'ha de recol·locar de fila o columna) o 2 (si s'ha de recol·locar de les dues). En el cas del rei, serà el màxim entre la seva posició actual i el nombre de files o columnes que estigui separat de la posició objectiu [2,4], ja que es pot moure en diagonal. Es retorna la suma dels dos valors.

Ens cal veure que aquesta és una heurística consistent (que implicarà que és també admisible). Una heurística és consistent si per tot node n, tot successor n' de n generat per qualsevol acció a es satisfà $h(n) \leq c(n, a, n') + h(n')$

Això és molt ràpid de provar. Inicialment $c(n, a, n') = 1 \ \forall n, a, n'$, ja que el cost de moure qualsevol peça en qualsevol direcció és 1. Després, $h(n') \geq h(n) - 1$ ja que de la manera de la qual hem definit l'heurística, com a molt la reduïrem en 1 a cada moviment (ja que, i com a molt, o bé l'heurística de la torre blanca es redueix en 1, o la del rei blanc es redueix en 1). Juntant-ho:

$$c(n, a, n') + h(n') = 1 + h'(n) \geq 1 + h(n) - 1 = h(n)$$

□

Podem extreure de l'anterior prova que $f(n)$ és no decreixent al llarg de qualsevol camí i que la cerca en grafs utilitzant l'algorisme A* és òptima.

Comentar finalment que aquesta és una heurística bona per al nostre cas concret (i que també serveix per bastants altres posicions inicials com s'explicarà en més detall al punt 2.4), però tenint si o si fixat el rei negre a la posició [0,4].

2.2 Implementació de l'algorisme A*

Tot i que el codi de l'algorisme comentat ja es troba a l'annex A, en aquest apartat especificarem encara més el seu funcionament i les decisions que hem pres en la seva implementació.

Per començar, inicialitzem la frontera com una cua de prioritat (ja que ens serà molt útil per després tenir-los ordenats per la seva funció $f(n)$) i hi insertem una tupla amb l'heurística del estat inicial i ell mateix (perquè

en l'estat inicial $f(n) = h(n)$ ja que $g(n) = 0$). Afegim l'estat inicial també a la llista d'estats visitats i al diccionari *dictpath*, el qual guarda de cada estat el seu predeccesor i la profunditat d'aquest (en l'estat inicial, indiquem que la profunditat del predecessor és -1). Inicialitzem també la variable *previousState* amb l'estat inicial, doncs no n'hi ha un realment.

Mentre la frontera no estigui buida (si ho estigués, es retornaria que ha fallat l'algorisme), extraiem de la frontera l'estat amb una $f(n)$ menor (fent simplement un *get*() ja que la frontera és una cua de prioritat). Fem una comprovació de no haver sobrepassat la màxima profunditat permesa i guardem en variables les dades del node que ens interessen (per fer un codi més entenedor).

A continuació, mirem quina peça s'ha mogut entre els dos estats i la movem realment en el taulell (això ho fem doncs sinó la funció $getListNextStatesW()$ no ens retorna tots els possibles estats). En el cas d'estar en un estat d'escac i mat (ho comprovem mitjançant la funció $isCheckMate()$), reconstruïm tot el camí, mostrem quina és la profunditat i acabem amb éxit l'execució de l'algorisme.

Si no és així, generem tots els possibles estats els quals es pugui accedir des de l'estat actual (i fem una comprovació ràpida per veure que l'ordre no ens doni problemes). Si no han sigut visitats anteriorment, es calcula el seu cost $f(n)$, com ja s'ha explicat al punt 2.1, i es posa a la frontera, a la llista dels estats visitats i es guarda la profunditat i l'estat predecessor a $dictPath$.

A més a més, tot i que no és necessari en aquest cas concret, per fer una implementació acurada de l'algorisme A*, si els estats ja han estat visitats però tenen un cost menor que el que tenien abans es reinserten en la frontera i s'actualitzen els valors en $dictPath$. Això, no només mai passarà en aquest cas concret, sinò que, a més, no ens cal extreure de la frontera l'anterior node que representava l'estat, doncs no arribarà tampoc mai a ser seleccionat.

2.3 Resultats obtinguts

Executant el codi amb les posicions de les peces: Rei negre = [0,4], Torre blanca = [7,0] i Rei blanc = [7,4] es mostra el següent resultat per pantalla, el qual és l'esperat i correcte:

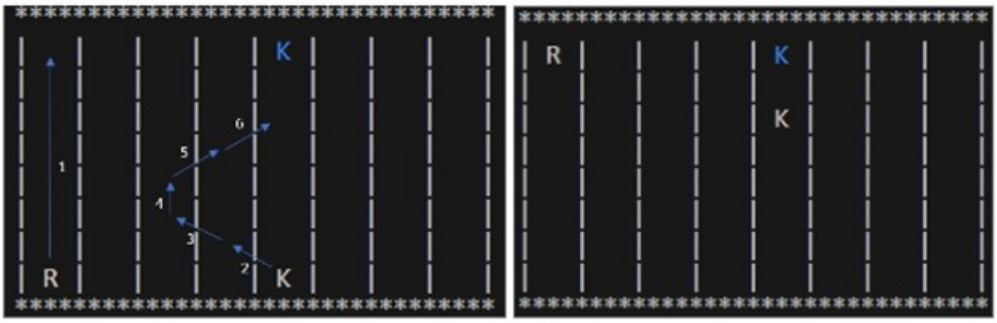


Figure 1: Representació de la seqüència de moviments per arribar a escat i mat amb el rei blanc a [7,4]

```
Output:
current State [[7, 0, 2], [7, 4, 6]]
A* minimal depth to reach target:  6
A* move sequence...   [[[7, 0, 2], [7, 4, 6]], [[0, 0, 2], [7, 4, 6]], [[0, 0, 2],
[6, 3, 6]], [[0, 0, 2], [5, 2, 6]], [[0, 0, 2], [4, 2, 6]], [[0, 0, 2], [3, 3, 6]],
[[0, 0, 2], [2, 4, 6]]]
A* End
```

2.4 Inicialitzant el taulell amb el rei blanc a la posició [7,7], funciona l'algorisme de la mateixa manera? Has hagut de canviar quelcom?

Executant de nou el codi, ara amb les posicions de les peces: Rei negre = [0,4], Torre blanca = [7,0] i Rei blanc = [7,7] es mostra el següent resultat per pantalla, el qual és l'esperat i correcte:

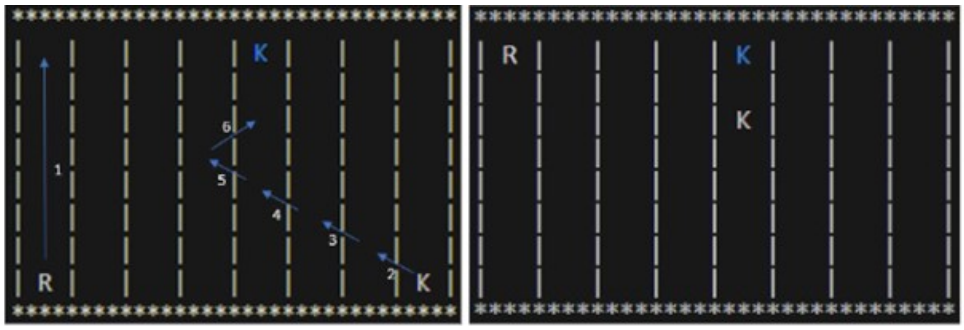


Figure 2: Representació de la seqüència de moviments per arribar a escat i mat amb el rei blanc a [7,7]

```
Output:
A* minimal depth to reach target:  6
A* move sequence...   [[[7, 0, 2], [7, 7, 6]], [[0, 0, 2], [7, 7, 6]], [[0, 0, 2],
[6, 6, 6]], [[0, 0, 2], [5, 5, 6]], [[0, 0, 2], [4, 4, 6]], [[0, 0, 2], [3, 3, 6]],
[[0, 0, 2], [2, 4, 6]]]
A* End
```

Així doncs l'algorisme funciona de la mateixa manera i no hem hagut de canviar res. Com hem vist abans en el punt 2.1, l'heurística que utilitzem no només és consistent, sinó que, a més, és independent de la posició inicial de les peces blanques, tot i que sí depén de quines són aquestes peces i de la posició estàtica del rei negre.

3 Conclusions i valoracions personals

Estem molts satisfets amb la feina que hem fet, doncs no només hem sigut capaços d'implementar amb èxit l'algorisme demanat, sinó que, a més, ens hem esforçat per optimitzar-lo, fer-lo net i entendible, així com hem procurat fer també en aquest informe.

Tot el temps invertit en la pràctica ens ha ajudat a entendre perfectament i assimilar l'algorisme A* i ens ha preparat, segurament, per afrontar les següents pràctiques de l'assignatura.

L'aspecte on hem trobat més dificultat o hem invertit més temps, a banda de corregir nombrosos errors a l'hora de programar, ha sigut en entendre la implementació de la resta del codi que ens venia donat. Inicialment, varem llegir els noms i que podíem fer amb cada métode, però, més endavant, va ser necessari per poder veure quins eren els nostres errors. Per exemple, ens va costar bastant veure que calia moure les peces del taulell perquè la funció $getListNextStatesW()$ ens retornés tots els possibles estats.

4 Contribució dels membres del grup

Per acabar aquest informe, comentar que hem treballat molt bé en equip, ens hem ajudat mútuament i ens hem repartit les tasques a fer de manera equitativa, sense alienar-nos en cap moment de la feina realitzada per l'altre company.

A Codi de l'algorisme A* comentat

```
def AStarSearch(self, currentState):

        #Inicialitzem la frontera, que sera una Priority Queue i li insertem el node
        #inicial amb la seva heuristica, ja que per aquest node inicial f(n) = h(n),
        #doncs g(n) = 0
        frontera = queue.PriorityQueue()
        frontera.put((self.h(currentState),currentState))

        #Insertem l'estat inicial a la llista dels visitats i en dictPath, on al no tenir
        #pare, hi insertem els valors que es poden veure
        self.listVisitedStates.append(currentState)
        self.dictPath[str(currentState)] = (None, -1)
        previousState = currentState

        #Mentre a la frontera hi quedin elements per visitar
        while frontera.not_empty:

            #Ens quedem amb el millor d'ells
            node = frontera.get()
            currentState = node[1]
            depthNode = self.dictPath[str(currentState)][1]+1

            #Comprovem que la profunditat no sigui mes gran que depthMax
            if(depthNode > self.depthMax):
                return False

            if depthNode > 0: # Si no es la primera iteracie
                previousState = self.dictPath[str(currentState)][0]

            #Mirem quina de les dues peces es mou
            if previousState[0] == currentState[0]:
                fitxaMoguda = 1 #S'ha mogut el rei
            else:
                fitxaMoguda = 0 #S'ha mogut la torre

            #Movem la taula per avancar amb la busqueda
            aichess.chess.moveSim(previousState[fitxaMoguda],currentState[fitxaMoguda])

            #Mirem si hem arribat a una posicio objectiu. En cas afirmatiu, reconstruim
            #el cami
            if(self.isCheckMate(currentState)):
              self.reconstructPath(currentState, depthNode)
              print("A* minimal depth to reach target: ", depthNode)
              break

           #Iterem sobre cada estat fill de l'estat actual
            for son in self.getListNextStatesW(currentState):

                #Fem una comprovacio d'ordre, per no tenir problemes mes endavant
                if son[0][2]==6:
                    son = [son[1],son[0]]

                #Calculem el cost estimat de la millor solucio que passa pel node
                cost_g_son = depthNode + 1
                cost_f_son = self.h(son) + cost_g_son

                #Si encara no haviem visitat l'estat o si el cost d'arribar al mateix node
                #per un altre cami es menor
                if (not self.isVisited(son)) or (depthNode < self.dictPath[str(son)][1]):

```

```
61                      #Afegim l'estat a la llista dels visitats i a la frontera. No ens
62                      #preocupem de tenir dos cops el mateix node, doncs l'antic, en cas
63                      #d'existir, no surtira mai de la frontera
64                      self.listVisitedStates.append(son)
65                      frontera.put((cost_f_son, son))
66                      self.dictPath[str(son)] = (currentState, depthNode)
67
68          #Si no queden nodes a la cua de prioritat vol dir que no hi ha solucio
69          return False
```