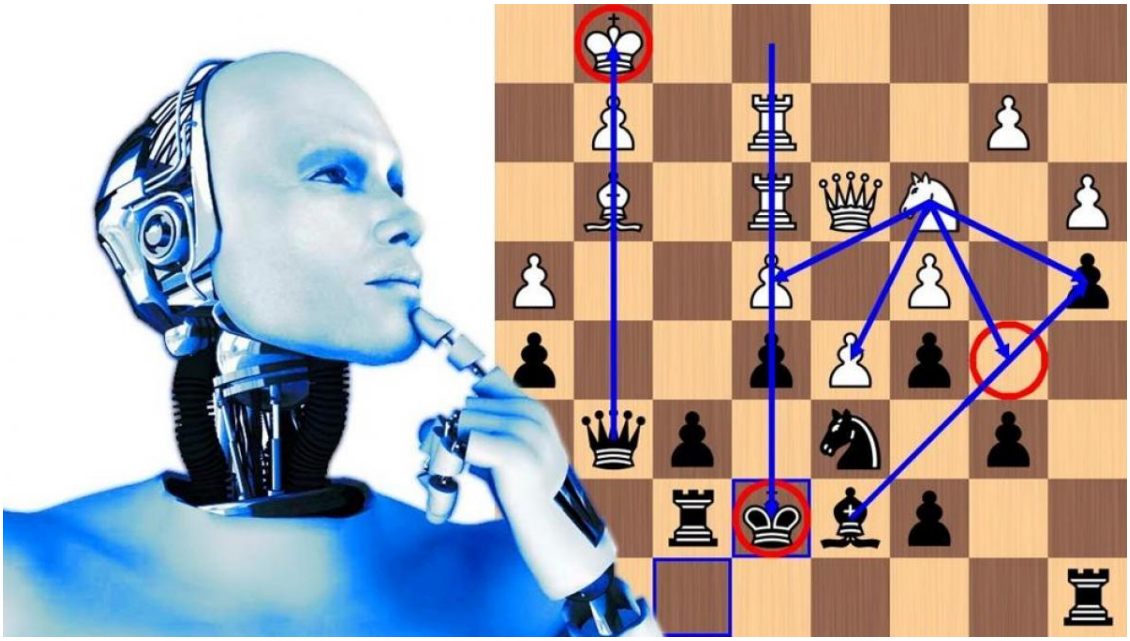
Universitat de Barcelona

Facultat de Matemàtiques i Informàtica

Pràctica 2: Adversarial Games

Intel·ligència Artificial



Carlos Bellanco Ortiz i Víctor Sort Rubio

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

Contents

1 Introducció

Els jocs amb oponents són una peça clau en el desenvolupament de la Intel·ligència Artificial. Aquests algoritmes representen un camp d'estudi essencial, ja que impliquen la creació d'estratègies competitives i adaptatives per afrontar escenaris dinàmics. En aquesta segona pràctica, seguint amb el mateix simulador del joc d'escacs que en la pràctica anterior, implementarem algorismes de jocs amb oponents per simular una partida reduïda.

Concretament, implementarem els algorismes minimax, minimax amb poda α-β i expectimax (juntament amb diverses altres funcions necessaries) per simular una partida d'escacs amb un rei blanc, un rei negre, una torre blanca i una torre negre. També analitzarem, variant i convinant els anteriors algorismes entre si, les sortides per pantalla que obtindrem.

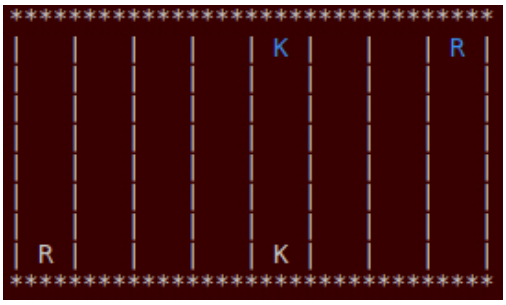


Figure 1: Representació de l'estat inicial de les peces.

2 Explicacions

Tot i que les funcions importants es troben ben comentades en els fitxers entregats, així com en l'annex d'aquest informe, hem trobat convenient fer una secció explicant com s'ha realitzat la implementació d'alguna funció o algorisme de manera general, així com alguna informació a tenir present.

2.1 Funció is_checkmate() i taules

És important considerar incialment que és impossible fer escac i mat al rival si només es disposa del rei, doncs aquest no es podrà apropar mai el suficient al rei adversari per amenaçar-lo, ja que ell mateix també es posaria "a tir".

Per simplificar el codi de la funció is_checkmate(), com seria simètric, està implementat sense tenir en compte el color, només les peces que els hi toca jugar, "atacants", i les que no, "defensants". Amb aquesta notació, òbviament, primer fem totes les inicialitzacions necessàries. Tenint present l'anterior consideració, si les peces atacants no tenen torre, es retorna fals.

Només estarem en situació d'escac i mat si el rei defensant està a les files o columnes 0 o 7. Mirem inicialment les columnes: La torre atacant ha d'estar a la mateixa, però no el costat del rei defensant. El rei atacant ha d'estar a dues columnes de separació i a la mateixa fila o una més, si es troba a una cantonada. Cal mirar també que la torre defensant no s'interposi en l'escac i mat, si es que encara està a la partida. El cas de les files serà 100% anàlog.

Expliquem a continuació les tres situacions on hem considerat a l'implementar els algorismes que el joc acaba amb empat, o més ben dit, amb taules:

1. Com ja hem justificat anteriorment, si les dues úniques peces vives es corresponen amb els dos reis.

2. Seguint les regles oficials dels escacs, si es produeixen 50 torns sense que cap equip es menji cap peça de l'equip rival.

3. Seguint també les regles oficials dels escacs, si un estat de la partida (entenent com a estat la posició de totes les peces del joc) es repeteix 3 vegades en una partida. Per implementar-ho al codi, hem creat un diccionari "dictVisitedStates" on les claus eren els estats visitats i els seus valors associats, el nombre de cops que ha succeït a la partida.

2.2 Implementació de l'algorisme Minimax

Per implementar l'algorisme minimax, primerament hem de definir els estats terminals. Un estat serà terminal quan la profunditat del node associat sigui igual a la profunditat màxima o quan es produeixi un escac i mat. Una vegada definits aquests estats, es comença la cerca del millor moviment possible sabent que el contrincant jugarà de forma òptima, és a dir, minimitzant la seva pèrdua. Per cada possible estat successor d'un estat, primer es comprova si amb aquest moviment s'ha produit un escac i mat i, si es així, al ser un estat terminal, retornem la seva heurística. Si no ha sigut escac i mat, comprovem que el moviment sigui legal, és a dir, que el rei no s'hagi posat en "check" del contrincant, i es continua amb la cerca donant el torn al contrincant.

A l'hora d'implementar l'algorisme, s'ha tingut en compte que si al realitzar un moviment es deixava al contrincant sense moviments possibles però aquest no estava en "check" (situació coneguda com rei ofegat), aquest moviment llavors era dolent ja que una partida on probablement s'hagués guanyat, acabaria en taules.

```
Algorithm 1: Minimax
  Data: Game description, node u, player player to act as MAX
  Result: The optimal move for player in node u's state
  (move, value) ← MAX-VALUE(u, player)
  return move

  function MAX-VALUE(u, player) :
      if terminal(u) then
          return (NULL, utility(u, player))
      end
      (best−move, max−value) ← (NULL, −∞)
      for max−move ∈ moves(u) do
          (min−move, value) ← MIN-VALUE(result(u, move), player)
          if value > max−value then
              (best−move, max−value) ← (max−move, value)
          end
      end
      return (best−move, max−value)
  end


  function MIN-VALUE(u, player) :
      if terminal(u) then
          return (NULL, utility(u, player))
      end
      (best−move, min−value) ← (NULL, +∞)
      for min−move ∈ moves(u) do
          (max−move, value) ← MAX-VALUE(result(u, move), player)
          if value < min−value then
              (best−move, min−value) ← (max−move, value)
          end
      end
      return (best−move, min−value)
  end
```

Figure 2: Pseudocodi de l'algorisme Minimax

2.3 Implementació de l'algorisme Minimax amb poda α-β

La implementació del minimax amb la poda α-β és exactament igual que amb el minimax però afegint els paràmetres α i β per tal de millorar considerablement el temps d'execució al realitzar podes per evitar expandir els estats que se sap que no donaran una millor solució. Així doncs, la implementació consisteix en una ràpida millora de l'anterior algorisme.

Es important recalcar que amb la poda α-β no es perd optimalitat ja que, com s'ha dit abans, es sap que aquests estats que no es visiten no donaran una millor solució.

2.4 Implementació de l'algorisme Expectimax

El funcionament d'aquest algorisme és molt semblant al del minimax però afegint després de cada moviment d'un jugador un node "chance" que calcula l'esperança d'aquest moviment donada una distribució de probabilitat i la utilitat dels estats successors. Els jugadors utilitzen el minimax per escollir el millor moviment de tal forma que, l'atacant triarà el moviment amb millor esperança i el defensant el de menor esperança per minimitzar la seva pèrdua. Hem usat per la implementar-la algunes funcions estadístiques que ja ens venien donades, com el càlcul de la mitjana i la desviació.

```
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
```

Figure 3: Pseudocodi de l'algorisme Poda α-β

Algorithm 2: Expectimax

Data: Game description, node u, player *player* to act as MAX
Result: The optimal move for *player* in node u's state
$(move, value) \leftarrow$ `EXPECTIMAX-VALUE`$(u,\ player)$
return *move*

function `EXPECTIMAX-VALUE`$(u,\ player)$ **:**
  if *u is a chance node* **then**
    $expected \leftarrow 0$
    for $r \in moves(u)$ **do**
      $(move, value) \leftarrow$ `EXPECTIMAX-VALUE`$(result(u, r),$ *player*$)$
      $expected \leftarrow expected + P(r) \cdot value$
    end
    return $(NULL, expected)$
  else if $turn(u) = player$ **then**
    return `MAX-VALUE`$(u,\ player)$
  else
    `// this is a MIN node`
    return `MIN-VALUE`$(u,\ player)$
  end
end

Figure 4: Pseudocodi de l'algorisme Expectimax

3 Respostes

3.1 Blanques: Minimax (profunditat 4) Negres: Minimax (profunditat 4)

3.1.1 Un cop implementat, fes córrer el mateix joc diverses vegades. Amb quina freqüència guanyen les blanques?

Hem fet córrer el codi 3 vegades (cada execució dura uns 7 minuts) i en totes s'obté el mateix resultat: Les blanques guanyen totes les vegades, amb una freqüència doncs del 100%, i l'estat final és sempre el mateix:

Figure 5: Estat final de les peces, amb l'algorisme minimax i profunditat 4-4.

A més, el camí d'estats "path to target" també és idèntic en totes les execucions:

```
[[[7, 0, 2], [7, 4, 6], [0, 4, 12], [0, 7, 8]], [[0, 0, 2], [7, 4, 6], [0, 4, 12], [0, 7, 8]],
[[0, 0, 2], [7, 4, 6], [1, 4, 12], [0, 7, 8]], [[0, 7, 2], [7, 4, 6], [1, 4, 12]],
[[0, 7, 2], [7, 4, 6], [2, 4, 12]], [[3, 7, 2], [7, 4, 6], [2, 4, 12]],
[[3, 7, 2], [7, 4, 6], [2, 3, 12]], [[3, 7, 2], [6, 3, 6], [2, 3, 12]],
[[3, 7, 2], [6, 3, 6], [2, 4, 12]], [[3, 7, 2], [5, 2, 6], [2, 4, 12]],
[[3, 7, 2], [5, 2, 6], [2, 3, 12]], [[3, 7, 2], [4, 2, 6], [2, 3, 12]],
[[3, 7, 2], [4, 2, 6], [2, 4, 12]], [[3, 7, 2], [4, 3, 6], [2, 4, 12]],
[[3, 7, 2], [4, 3, 6], [2, 5, 12]], [[3, 7, 2], [4, 4, 6], [2, 5, 12]],
[[3, 7, 2], [4, 4, 6], [2, 4, 12]], [[2, 7, 2], [4, 4, 6], [2, 4, 12]],
[[2, 7, 2], [4, 4, 6], [1, 3, 12]], [[2, 7, 2], [4, 3, 6], [1, 3, 12]],
[[2, 7, 2], [4, 3, 6], [1, 2, 12]], [[2, 7, 2], [3, 3, 6], [1, 2, 12]],
[[2, 7, 2], [3, 3, 6], [1, 3, 12]], [[1, 7, 2], [3, 3, 6], [1, 3, 12]],
[[1, 7, 2], [3, 3, 6], [0, 2, 12]], [[1, 7, 2], [2, 3, 6], [0, 2, 12]],
[[1, 7, 2], [2, 3, 6], [0, 3, 12]], [[0, 7, 2], [2, 3, 6], [0, 3, 12]]]
```

3.1.2 Proporciona una justificació.

Al veure que totes les sortides són idèntiques podem deduïr que així serà també si seguíssim executant el codi indeterminadament. De fet, com cada cop que s'executa el codi es parteix del mateix estat inicial, la seqüència d'estats que es generarà serà sempre la mateixa i aquesta només canviaria si es canvien els valors de les profunditats passats per paràmetre.

Podem veure que el moviment inicial més òptim a fer per part de les blanques és moure la torre de la posició [7,0] a la [0,0]. D'aquesta manera, es fa escac al rei negre que es troba a la posició [0,4] i aquest ha d'abandonar la fila 0, passant-se a la fila 1. Així doncs, la torre blanca podrà menjar-se a la torre negra que es troba a la posició [0,7] fent que, segons ja hem explicat anteriorment, le peces negres no puguin guanyar mai. A partir d'aquest punt, com el rei negre no es pot menjar immediatament la torre blanca, és qüestió de temps que les blanques guanyin o es produeixin taules, si succeeix un dels casos explicats a l'apartat 2.1.

3.2 Blanques: Minimax (profunditat 1-4) Negres: Minimax (profunditat 1-4)

3.2.1 Traça el percentatge de victòries de les blanques sobre el total per a cada valor de profunditat.

BLACK / WHITE	DEPTH 1	DEPTH 2	DEPTH 3	DEPTH 4
DEPTH 1	100% TAULES	100% TAULES	100% TAULES	100% TAULES
DEPTH 2	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES
DEPTH 3	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES
DEPTH 4	100% BLANQUES	100% BLANQUES	100% BLANQUES	100% BLANQUES

Figure 6: Taula indicant els resultats, amb l'algorisme minimax i profunditats variants.

Clarament, per dos donats valors de profunditat, el resultat sempre serà el mateix, com hem vist abans. Això justifica que sempre es tingui un 100% de victòries de les blanques o un 100% de taules.

3.2.2 És el resultat simètric? Per què?

Totes les partides, independentment de la profunditat, comencen de la mateixa manera que hem explicat a l'apartat 3.1.2, fent que la torre blanca es menji la torre negre i impedint que les peces negres puguin guanyar mai. Això justifica que el resultat final sigui o taules o victòria per les blanques.

S'observa una simetria clara que es podria modelitzar de la següent manera:

$$Resultats = f(depth\_black, depth\_black) = \begin{cases} 100\% Blanques & si \quad depth\_black < depth\_white \\ \\ 100\% Taules & si \quad depth\_black > depth\_white \end{cases}$$

Pel cas $depth\_black = depth\_white$, el resultat varia depenent de la profunditat.

La justificació d'aquesta simetria és que si la profunditat de les peces negres és superior a les de les peces blanques, aquestes s'anticiparan a tots els moviments que puguin fer, obligant a les blanques a fer moviments que ja saben que arribaran a un bucle infinit i la partida acabarà amb taules. De manera anàloga serà si la profunditat de les blanques és superior a la de les negres, les blanques veuran més enllà que les negres i podran escollir un moviment sabent que faran les peces negres amb seguretat. Si les profunditats són iguals el resultat no queda determinat ja que cap equip veurà més del que l'altre pot mai.

3.3 Blanques: Minimax (profunditat 4) Negres: Poda α-β (profunditat 4)

3.3.1 Fes córrer la simulació tres vegades. Qui és el millor de les tres?

Obtenim exactament els mateixos resultats que usant l'algorisme minimax pels dos equips, vist a l'apartat 3.1. La diferéncia radica en que el temps d'execució es redueix a la meitat, rondant els 4 minuts. Així doncs, les peces blanques guanyen el 100% de les vegades i es té el següent estat final:

Figure 7: Estat final de les peces, amb l'algorisme minimax amb i sense poda α-β i profunditat 4-4.

3.3.2 Per què és així?

Que totes les simulacions tinguin el mateix resultat entre elles és per el mateix motiu que hem vist en anterioritat: els arbres d'estats generats cada vegada són idèntics.

Per respondre a perquè s'obté el mateix resultat que aplicant només l'algorisme minimax sense poda α-β cal entendre com funciona l'algorisme amb poda α-β. Com bé s'explica en el punt 2.3, aquest no és més que una millora del minimax on s'evita expandir l'arbre d'estats si es sap que no trobarem un resultat millor del que ja tenim. Per això, l'estat final (i de fet, tots els estats que conformen el "path to target") és idèntic al de l'apartat 3.1.1. L'única diferència és, com ja hem mencionat, una reducció del temps d'execució, que és la finalitat d'aplicar la poda α-β a l'algorisme minimax.

3.4 Blanques: Poda α-β (profunditat 1-5) Negres: Poda α-β (profunditat 1-5)

3.4.1 Traça la proporció de victòries per les blanques sobre les negres.

Tal i com hem fet en l'apartat 3.2.1, obtenim la següent taula:

BLACK / WHITE	DEPTH 1	DEPTH 2	DEPTH 3	DEPTH 4	DEPTH 5
DEPTH 1	100% TAULES	100% TAULES	100% TAULES	100% TAULES	100% TAULES
DEPTH 2	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES	100% TAULES
DEPTH 3	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES	100% TAULES
DEPTH 4	100% BLANQUES	100% BLANQUES	100% BLANQUES	100% BLANQUES	100% BLANQUES
DEPTH 5	100% TAULES	100% TAULES	100% TAULES	100% TAULES	100% TAULES

Figure 8: Taula indicant els resultats, amb l'algorisme minimax amb poda α-β i profunditats variants.

3.4.2 Comenta el resultat.

Per profunditats entre 1 i 4, s'obté la mateixa taula que a l'apartat 3.2.1, que és obvi doncs es tracta del mateix algorisme. Com ja hem comentat, a l'implementar la millora de la poda α-β el temps de l'execució es redueix permetent-nos veure altres combinacions superiors. Com sempre, les negres mai guanyen per la desigualtat creada en els primers moviments. Podria sobtar-nos algunes combinacions noves que veiem, per exemple que amb profunditats respectives de blanques i negres 5-1 s'acabi en taules si amb profunditats 4-1 guanyen les blanques. Cal tenir present però que els algorisme es regeixen per una heurística, la qual no és perfecta (per exemple, no sap considerar quan estem en una situació de taules, on hauria de retornar un valor de 0). És per aquest motiu que tot i ser la profunditat de les blanques superior a la de les negres, la partida pot acabar en taules igualment.

3.5 Blanques: Expectimax (profunditat 1-4) Negres: Poda α-β (profunditat 1-4)

3.5.1 Fes córrer tres simulacions cadascuna i traça la proporció de victòries per les blanques sobre les negres.

Variant les profunditats, obtenim la següent taula:

BLACK / WHITE	DEPTH 1	DEPTH 2	DEPTH 3	DEPTH 4
DEPTH 1	100% TAULES	100% TAULES	100% TAULES	100% TAULES
DEPTH 2	100% TAULES	100% TAULES	100% TAULES	100% TAULES
DEPTH 3	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES
DEPTH 4	100% BLANQUES	100% BLANQUES	100% TAULES	100% TAULES

Figure 9: Taula indicant els resultats, amb l'algorisme expectimax i profunditats variants.

3.5.2 Qui guanya més sovint? Per què és així?

L'avantatge que tenen les blanques envers les negres fa que les negres no guanyin mai, com ja hem comentat diverses vegades. Ara, comparat amb la taula de l'apartat 3.2.1 es veu que el nombre de victòries de les blanques es redueix i es converteixen en taules. Això és així perqué amb aquest algorisme les peces blanques tenen en compte tots els possibles escenaris que poden fer les blanques i calculen la mitja de l'heurística, convertint la jugada que escullen en "més conservadora" o dit d'una altre manera, la que minimitzi el seu risc de perdre. Es suposa doncs en l'algorisme que no es té la certesa de que farà l'altra equip.

3.6 Preguntes finals

3.6.1 La situació generada enfrontant un rei blanc i un rei negre més una torre cadascun pot ser considerada com a justa. És realment així? Justifica la teva resposta.

No és cert que una partida d'escacs on les peces blanques i les negres disposin de un rei i una torre sigui una partida 100% justa per dos motius:

1. Posició inicial de les peces: Si es tingués una configuració inicial on amb un sol moviment les blanques féssin escac i mat a les negres és obvi que no estem en una situació justa, doncs el 100% de les partides les guanyarien les blanques sense les negres tenir cap possibilitat ni de moure's. És clar doncs que la posició inicial és molt important per poder predir qui serà el guanyador.

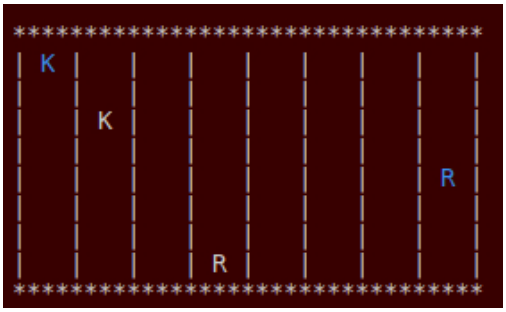


Figure 10: Exemple de taulell on les blanques guanyarien el 100% de les vegades amb un sol moviment.

2. Avantatge del jugador inicial: Al ser les blanques les que comencen la partida, les negres aniran un torn endarrere a les blanques quan les blanques moguin, i amb el mateix nombre de torns quan les negres moguin. Aquestes diferències puntuals d'un torn produeixen un avantatge a les peces blanques ja de per si en una partida normal (la probabilitat estadística de guanyar jugant amb blanques ronda el 38% i jugant amb negres el 32%). En la posició inicial de les peces amb les que partim, aquesta diferència s'accentua encara més proporcionant un avantatge molt superior a les peces blanques. Al ser les primeres en jugar, al moure la torre blanca de la posició [7,0] a la posició [0,0] ja es posen amb una gran avantatge respecte les negres, com hem anat veient durant tota la pràctica.

En resum, amb les característiques donades no es pot afirmar que la situació generada sigui justa.

3.6.2 En la teva opinió, què fa aquesta situació particularment interessant per a l'estudi dels jocs adversarials?

La situació que se'ns proposa és una gran manera per estudiar els jocs amb adversaris per els següents motius:

1. Incialment, té una complexitat estratègica important, doncs per decidir quina és la millor jugada a fer a cada torn cal pensar en quin serà l'estat de la partida a continuació, els possibles estats o jugades que es poden desencadenar i fer per tant una planificació a llarg termini. Amb els algorismes estudiats aquest concepte de "llarg termini" es veu representat pel concepte de "profunditat de l'algorisme". És doncs un cas ideal per poder aplicar aquests algorismes estudiats a teoria.

2. A més, a l'haver poques peces és senzill seguir mentalment la dinàmica del joc i poder veure i pensar de manera directa les possibles jugades a fer i com afectaran a la partida. Així doncs es pot comprovar mentalment el correcte funcionament dels algorismes implementats i la seva comprensió es facilita.

4 Conclusions i valoracions personals

Estem molts satisfets amb la feina que hem fet, doncs no només hem sigut capaços d'implementar amb èxit els tres algorismes demanats, sinó que, a més, ens hem esforçat per optimitzar-los i fer-los nets i entenedors, així com hem procurat fer també en aquest informe.

Tot el temps invertit en la pràctica ens ha ajudat a entendre perfectament i assimilar els algorismes de jocs amb oponents vistos a teoria i ens ha semblat una molt bona manera d'aplicar els conceptes apresos, a més d'entretinguda i motivadora.

L'aspecte que més ens ha costat ha sigut tractar els bucles que convertien els moviments del taulell en cicles. Vem estar diversa estona intentant evitar-los, fins que vàrem decicir basar-nos en les regles oficials dels escacs. Tot el procés plegat va suposar bastant de temps una mica tediós doncs vam haver de canviar gran part de la implementació.

El temps dedicat a la pràctica ha sigut bastant quantiós i agraïm que s'aplaçés l'entrega de la mateixa un cop finalitzats els examens parcials, doncs sinó no haguéssim tingut el temps d'entregar una pràctica amb la qualitat que té actualment.

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 les funció is_checkmate() comentat

```
def is_Checkmate(self,currentState,color):
    #Per no duplicar el codi, ho farem sense tenir en compte el color, nomes a la
    #inicialitzacio
    #A = el que li toca jugar, "ataca"   D = el que no li toca jugar, "defensa"

    #Inicialitzem totes les variables
    if color:
        dkState = self.getPieceState(currentState, 12)
        akState = self.getPieceState(currentState, 6)
        arState = self.getPieceState(currentState, 2)
        drState = self.getPieceState(currentState, 8)
    else:
        akState = self.getPieceState(currentState, 12)
        dkState = self.getPieceState(currentState, 6)
        drState = self.getPieceState(currentState, 2)
        arState = self.getPieceState(currentState, 8)

    filaAk = akState[0]
    columnaAk = akState[1]
    filaDk = dkState[0]
    columnaDk = dkState[1]

    if arState != None:
        filaAr = arState[0]
        columnaAr = arState[1]
    else:
        return False #No es possible conseguir checkMate amb nomes el rei

    if drState != None:
        filaDr = drState[0]
        columnaDr = drState[1]

    #Nomes estarem en situacio d'escac i mat si el rei D esta a les files o columnes 0 o 7
    #Mirem inicialment les columnes
    if (columnaDk == 0) or (columnaDk == 7):
        #La torre A ha d'estar a la mateixa, pero no el costat del rei D
        if (columnaDk == columnaAr) and (abs(filaDk - filaAr) > 1):
            #El rei A ha d'estar a dues columnes de separacio i a la mateixa fila o una mes,
            #si es troba a una cantonada
            if abs(columnaDk - columnaAk) == 2 and ((filaDk == filaAk) or (abs(filaDk -
            filaAk) == 1 and (filaDk == 0 or filaDk == 7))):
                if drState == None:
                    return True
                if(filaAk != filaDr and columnaAk != columnaDr):
                    if(filaAk == filaDk):
                        if(filaAr < filaDk and (filaDr < filaAr or filaDr > filaDk)) or
                        (filaAr > filaDk and (filaDr > filaAr or filaDr < filaDk)):
                            return True
                        #Mirem aqui les quatre configuracions diferents (rei que es defensa a
                        #una cantonada i al que ataca a una fila diferent)
                        if(filaAr < filaDk and (filaDr < filaAr or filaDr == filaDk)) or
                        (filaAr > filaDk and (filaDr > filaAr or filaDr == filaDk)):
                            return True

    #De manera analoga mirarem les files
    if (filaDk == 0) or (filaDk == 7):
        if (filaDk == filaAr and abs(columnaDk - columnaAr) > 1) or ((abs(columnaDk -
        columnaAk) == 1 and (columnaDk == 0 or columnaDk == 7))):
            if (abs(filaDk - filaAk) == 2 and columnaDk == columnaAk):
                if drState == None:
```

```
                    return True
                if(columnaAk == columnaDk)
                    if(filaAk != filaDr and columnaAk != columnaDr):
                        if(columnaAr < columnaDk and (columnaDr < columnaAr or columnaDr
                        > columnaDk)) or (columnaAr > columnaDk and (columnaDr > columnaAr
                        or columnaDr < columnaDk)):
                            return True
                        if(columnaAr < columnaDk and (columnaDr < columnaAr or columnaDr ==
                        columnaDk)) or (columnaAr > columnaDk and (columnaDr > columnaAr or
                        columnaDr == columnaDk)):
                            return True

    #Si no estem en cap situacio anterior, retornem false
    return False
```

B Codis de la implementació de l'algorisme Minimax comentats

```
def minimaxGame(self, depthWhite,depthBlack):
    currentState = self.getCurrentState()
    currentStateTuple = tuple(tuple(row) for row in currentState)
    color = True #Primer mouen les blanques
    movSenseMenjar = 0 #Si arriba a 50, son taules
    nombre_fitxes = len(currentState)
    ciclic = False
    self.dictVisitedStates = {currentStateTuple : 1} #Si el valor d'alguna clau arriba a 3,
    #son taules
    self.pathToTarget.append(copy.deepcopy(currentState))
    self.minimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())

    #A cada torn anem cridant la funcio minimaxDecision i mirem si es escac i mat o taules
    while (not self.is_Checkmate(self.getCurrentState(), color)) and
    len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:
        currentState = self.getCurrentState()
        currentStateTuple = tuple(tuple(row) for row in currentState)

        self.pathToTarget.append(copy.deepcopy(currentState))

        #Analitzem si s'ha menjat una peca
        if(len(currentState) < nombre_fitxes):
            nombre_fitxes = len(currentState)
            movSenseMenjar = 0
        else:
            movSenseMenjar += 1

        #Actualitzem el diccionari d'estats visitats
        if currentStateTuple in self.dictVisitedStates.keys():
            self.dictVisitedStates[currentStateTuple] += 1
            if self.dictVisitedStates[currentStateTuple] == 3:
                ciclic = True
        else:
            self.dictVisitedStates[currentStateTuple] = 1

        color = not color
        aichess.chess.board.print_board()
        self.minimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())

    currentState = self.getCurrentState()
    self.pathToTarget.append(copy.deepcopy(currentState))

    #Mostrem per pantalla el resultat
```

```
    if self.is_Checkmate(currentState, color):
        if color:
           print("Guanyen les blanques!")
        else:
            print("Guanyen les negres!")
    else:
        print("Taules!")

    #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
    aichess.chess.board.print_board()


def minimaxDecision(self, depthWhite, depthBlack, color, currentState):
    if(color):
        depth = depthWhite
    else:
        depth = depthBlack

    self.max_value(0,color,currentState,depth)


def max_value(self, currentDepth, color, currentState, depthMax):
    if(currentDepth == depthMax): #Estat terminal
        return self.heuristica(currentState, color)

    v=float("-inf")

    #Creem un nou tauler amb l'estat actual
    self.newBoardSim(currentState)

    if(color):
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getWhiteState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                if(currentDepth == 0):
                    nextMove = movement
                    aichess.chess.move(nextMove[0],nextMove[1])
                    self.newBoardSim(self.getCurrentState())
                return self.heuristica(self.getCurrentSimState(), color)

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les negres
            if not(self.isWatchedWk(self.getCurrentSimState())):
                s = max(v,self.min_value(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v < s):
                        nextMove = movement
                v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    else:
        #Mirem tots els seguents estats possibles
```

```
        for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getBlackState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                if(currentDepth == 0):
                    nextMove = movement
                    aichess.chess.move(nextMove[0],nextMove[1])
                        self.newBoardSim(self.getCurrentState())
                return self.heuristica(self.getCurrentSimState(), color)

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les blanques
            if not(self.isWatchedBk(self.getCurrentSimState())):
                s = max(v,self.min_value(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v < s):
                        nextMove = movement
                v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    #Una vegada trobat el millor moviment, el fem al tauler real
    if(currentDepth == 0):
        aichess.chess.move(nextMove[0],nextMove[1])
        self.newBoardSim(self.getCurrentState())

    return v


def min_value(self, currentDepth, color, currentState, depthMax):
    if(currentDepth == depthMax): #Estat terminal
        return self.heuristica(currentState, not color)

    v=float("inf")

    #Creem un nou tauler amb l'estat actual
    self.newBoardSim(currentState)

    if(color):
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getWhiteState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les negres
            if not(self.isWatchedWk(self.getCurrentSimState())):
                s = min(v,self.max_value(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
```

```
                if(currentDepth == 0):
                    if(v > s):
                        nextMove = movement
                v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    else:
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesB(self.getBlackState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getBlackState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les blanques
            if not(self.isWatchedBk(self.getCurrentSimState())):
                s = min(v,self.max_value(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v > s):
                        nextMove = movement
                v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    #Si aixo passa, vol dir que no es pot fer cap moviment legal i, per tant, el moviment
    #es dolent
    if(v==float('inf')):
        v = float('-inf')

    return v
```

C Codis de la implementació de l'algorisme Minimax amb poda α-β comentats

```
def alphaBetaPodaGame(self, depthWhite,depthBlack):

    currentState = self.getCurrentState()
    currentStateTuple = tuple(tuple(row) for row in currentState)
    color = True; #Primer mouen les blanques
    movSenseMenjar = 0
    nombre_fitxes = len(currentState)
    ciclic = False
    self.dictVisitedStates = {currentStateTuple : 1}
    self.pathToTarget.append(copy.deepcopy(currentState))
    self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())

    #A cada torn anem cridant la funcio alphaBetaPodaDecision i mirem si es escac i mat
    #o taules
    while (not self.is_Checkmate(self.getCurrentState(), color)) and
    len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:
```

```
        currentState = self.getCurrentState()
        currentStateTuple = tuple(tuple(row) for row in currentState)
        self.pathToTarget.append(copy.deepcopy(currentState))

        #Analitzem si s'ha menjat una peca
        if(len(currentState) < nombre_fitxes):
            nombre_fitxes = len(currentState)
            movSenseMenjar = 0
        else:
            movSenseMenjar += 1

        #Actualitzem el diccionari d'estats visitats
        if currentStateTuple in self.dictVisitedStates.keys():
            self.dictVisitedStates[currentStateTuple] += 1
            if self.dictVisitedStates[currentStateTuple] == 3:
                ciclic = True
        else:
            self.dictVisitedStates[currentStateTuple] = 1

        color = not color
        aichess.chess.board.print_board()
        self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())

    self.pathToTarget.append(copy.deepcopy(currentState))

    #Mostrem per pantalla el resultat
    if self.is_Checkmate(self.getCurrentState(), color):
        if color:
            print("Guanyen les blanques!")
        else:
            print("Guanyen les negres!")
    else:
        print("Taules!")

    #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
    aichess.chess.board.print_board()


def alphaBetaPodaDecision(self, depthWhite, depthBlack, color, currentState):
    if(color):
        depth = depthWhite
    else:
        depth = depthBlack

    self.alphaBetaMax(0,color,currentState,depth,float('-inf'),float('inf'))


def alphaBetaMax(self, currentDepth, color, currentState, depthMax, alpha, beta):
    if(currentDepth == depthMax): #Estat terminal
        return self.heuristica(currentState, color)

    v=float("-inf")

    #Creem un nou tauler amb l'estat actual
    self.newBoardSim(currentState)

    if(color):
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getWhiteState(currentState),nextState)
```

```
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                if(currentDepth == 0):
                    nextMove = movement
                    aichess.chess.move(nextMove[0],nextMove[1])
                    self.newBoardSim(self.getCurrentState())
                return self.heuristica(self.getCurrentSimState(), color)

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les negres
            if not(self.isWatchedWk(self.getCurrentSimState())):
                s = max(v,self.alphaBetaMin(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax,alpha,beta))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v < s):
                        nextMove = movement
                v = s
                if v >= beta:
                    return v
                alpha = max(alpha, v)

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    else:
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getBlackState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                if(currentDepth == 0):
                    nextMove = movement
                    aichess.chess.move(nextMove[0],nextMove[1])
                    self.newBoardSim(self.getCurrentState())
                return self.heuristica(self.getCurrentSimState(), color)

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les blanques
            if not(self.isWatchedBk(self.getCurrentSimState())):
                s = max(v,self.alphaBetaMin(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax,alpha,beta))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v < s):
                        nextMove = movement
                v = s
                if v >= beta:
                    return v
                alpha = max(alpha, v)

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    #Una vegada trobat el millor moviment, el fem al tauler real
    if(currentDepth == 0):
        aichess.chess.move(nextMove[0],nextMove[1])
```

```
        self.newBoardSim(self.getCurrentState())

    return v


def alphaBetaMin(self, currentDepth, color, currentState, depthMax, alpha, beta):
    if(currentDepth == depthMax): #Estat terminal
        return self.heuristica(currentState, not color)

    v=float("inf")

    #Creem un nou tauler amb l'estat actual
    self.newBoardSim(currentState)

    if(color):
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getWhiteState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les negres
            if not(self.isWatchedWk(self.getCurrentSimState())):
                s = min(v,self.alphaBetaMax(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax,alpha,beta))
                #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
                if(currentDepth == 0):
                    if(v > s):
                        nextMove = movement
                v = s
                if v <= alpha:
                    return v
                beta = min(beta, v)

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    else:

        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesB(self.getBlackState(currentState)):

            #Mirem el moviment que es produeix i el fem
            movement = self.getMovement(self.getBlackState(currentState),nextState)
            aichess.chess.moveSim(movement[0],movement[1],False)

            #Comprovem si l'estat actual es checkmate
            if(self.is_Checkmate(self.getCurrentSimState(), color)):
                return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal

            #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
            #les blanques
            if not(self.isWatchedBk(self.getCurrentSimState())):
                s = min(v,self.alphaBetaMax(currentDepth+1,not color,
                self.getCurrentSimState(),depthMax,alpha,beta))
                #Si estem a l'estat inicial, guardem el moviment amb millor heurIstica
```

```
                if(currentDepth == 0):
                    if(v > s):
                        nextMove = movement
                v = s
                if v <= alpha:
                    return v
                beta = min(beta, v)

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    if(currentDepth == 0):
        aichess.chess.move(nextMove[0],nextMove[1])
        self.newBoardSim(self.getCurrentState())

    #Si aixo passa, vol dir que no es pot fer cap moviment legal i, per tant, el moviment
    #es dolent
    if(v==float('inf')):
        v = float('-inf')

    return v
```

D Codis de la implementació de l'algorisme Expectimax comentats

```
def expectimaxGame(self, depthWhite, depthBlack):
    currentState = self.getCurrentState()
    currentStateTuple = tuple(tuple(row) for row in currentState)
    color = True; #Primer mouen les blanques
    movSenseMenjar = 0
    nombre_fitxes = len(currentState)
    ciclic = False
    self.dictVisitedStates = {currentStateTuple : 1}
    self.pathToTarget.append(copy.deepcopy(currentState))
    self.expectimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())

    #A cada torn anem cridant la funcio minimaxDecision i mirem si es escac i mat o taules
    while (not self.is_Checkmate(self.getCurrentState(), color))
    and len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:
        currentState = self.getCurrentState()
        currentStateTuple = tuple(tuple(row) for row in currentState)
        self.pathToTarget.append(copy.deepcopy(currentState))

        #Analitzem si s'ha menjat una peca
        if(len(currentState) < nombre_fitxes):
            nombre_fitxes = len(currentState)
            movSenseMenjar = 0
        else:
            movSenseMenjar += 1

        #Actualitzem el diccionari d'estats visitats
        if currentStateTuple in self.dictVisitedStates.keys():
            self.dictVisitedStates[currentStateTuple] += 1
            if self.dictVisitedStates[currentStateTuple] == 3:
                ciclic = True
        else:
            self.dictVisitedStates[currentStateTuple] = 1

        color = not color
        aichess.chess.board.print_board()

```

```
        if(color):
            self.expectimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())
        else:
            self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())

    currentState = self.getCurrentState()
    self.pathToTarget.append(copy.deepcopy(currentState))

    #Mostrem per pantalla el resultat
    if self.is_Checkmate(currentState, color):
        if color:
            print("Guanyen les blanques!")
        else:
            print("Guanyen les negres!")
    else:
        print("Taules!")

    #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
    aichess.chess.board.print_board()

def expectimaxDecision(self, depthWhite, depthBlack, color, currentState):
    if(color):
        depth = depthWhite
    else:
        depth = depthBlack

    self.expectimax(0,color,currentState,depth,"max","None")

def expectimax(self, currentDepth, color, currentState, depthMax, nodeType,
    previousNodeType):
    nextMove = []
    values = []
    if(currentDepth == depthMax): #Estat terminal
        if(nodeType == "min" or previousNodeType == "max"):
            return self.heuristica(currentState, not color)
        return self.heuristica(currentState, color)

    if(nodeType == "max"):
        v=float('-inf')
    else:
        v=float('inf')

    #Creem un nou tauler amb l'estat actual
    self.newBoardSim(currentState)

    if(color):

        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
            #Si estem a un node chance, calculem el valor de cada estat successor i l'afegim
            #a la llista values
            if(nodeType == "chance"):
                #Si el node anterior era max, el seguent sera min i viceversa
                if(previousNodeType == "max"):
                    values.append(self.expectimax(currentDepth+1, color,
                    self.getCurrentSimState(), depthMax, nodeType="min",
                    previousNodeType="chance"))
                elif(previousNodeType == "min"):
                    values.append(self.expectimax(currentDepth+1, color,
                    self.getCurrentSimState(), depthMax, nodeType="max",
                    previousNodeType="chance"))

```

```
            #Si estem a un node min o max
            else:
                #Mirem el moviment que es produeix i el fem
                movement = self.getMovement(self.getWhiteState(currentState),nextState)
                aichess.chess.moveSim(movement[0],movement[1],False)

                #Comprovem si l'estat actual es checkmate
                if(self.is_Checkmate(self.getCurrentSimState(), color)):
                    if(currentDepth == 0):
                        nextMove = movement
                        aichess.chess.move(nextMove[0],nextMove[1])
                        self.newBoardSim(self.getCurrentState())
                    return self.heuristica(self.getCurrentSimState(), color)

                #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check
                #de les negres
                if not(self.isWatchedWk(self.getCurrentSimState())):
                    #Calculem el maxim o minim dels nodes chance successors
                    if(nodeType == "max"):
                        s = max(v,self.expectimax(currentDepth+1, not color,
                        self.getCurrentSimState(), depthMax, nodeType="chance",
                        previousNodeType="max"))
                    elif(nodeType == "min"):
                        s = min(v, self.expectimax(currentDepth+1, not color,
                        self.getCurrentSimState(), depthMax, nodeType="chance",
                        previousNodeType="min"))

                    #Si estem a l'estat inicial, guardem el moviment amb millor esperanca
                    if(currentDepth == 0):
                        if(v < s):
                            nextMove = movement
                    v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    else:
        #Mirem tots els seguents estats possibles
        for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
            #Si estem a un node chance, calculem el valor de cada estat successor i l'afegim
            #a la llista values
            if(nodeType == "chance"):
                #Si el node anterior era max, el seguent sera min i viceversa
                if(previousNodeType == "max"):
                    values.append(self.expectimax(currentDepth+1, color,
                    self.getCurrentSimState(), depthMax, nodeType="min",
                    previousNodeType="chance"))
                elif(previousNodeType == "min"):
                    values.append(self.expectimax(currentDepth+1, color,
                    self.getCurrentSimState(), depthMax, nodeType="max",
                    previousNodeType="chance"))
            #Si estem a un node min o max
            else:
                #Mirem el moviment que es produeix i el fem
                movement = self.getMovement(self.getBlackState(currentState),nextState)
                aichess.chess.moveSim(movement[0],movement[1],False)

                #Comprovem si l'estat actual es checkmate

                if(self.is_Checkmate(self.getCurrentSimState(), color)):
                    if(currentDepth == 0):
                        nextMove = movement
```

```
                        aichess.chess.move(nextMove[0],nextMove[1])
                        self.newBoardSim(self.getCurrentState())
                    return self.heuristica(self.getCurrentSimState(), color)

                #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check
                #de les blanques
                if not(self.isWatchedBk(self.getCurrentSimState())):
                    #Calculem el maxim o minim dels nodes chance successors
                    if(nodeType == "max"):
                        s = max(v,self.expectimax(currentDepth+1, not color,
                        self.getCurrentSimState(), depthMax, nodeType="chance",
                        previousNodeType="max"))
                    elif(nodeType == "min"):
                        s = min(v, self.expectimax(currentDepth+1, not color,
                        self.getCurrentSimState(), depthMax, nodeType="chance",
                        previousNodeType="min"))

                    #Si estem a l'estat inicial, guardem el moviment amb millor esperanca
                    if(currentDepth == 0):
                        if(v < s):
                            nextMove = movement
                    v = s

            #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
            self.newBoardSim(currentState)

    #Una vegada coneguts tots els valors dels estats successors calculem l'esperanca del
    #node chance
    if(nodeType == "chance"):
        v = self.calculateValue(values)

    if(currentDepth == 0):
        aichess.chess.move(nextMove[0],nextMove[1])
        self.newBoardSim(self.getCurrentState())

    return v
```