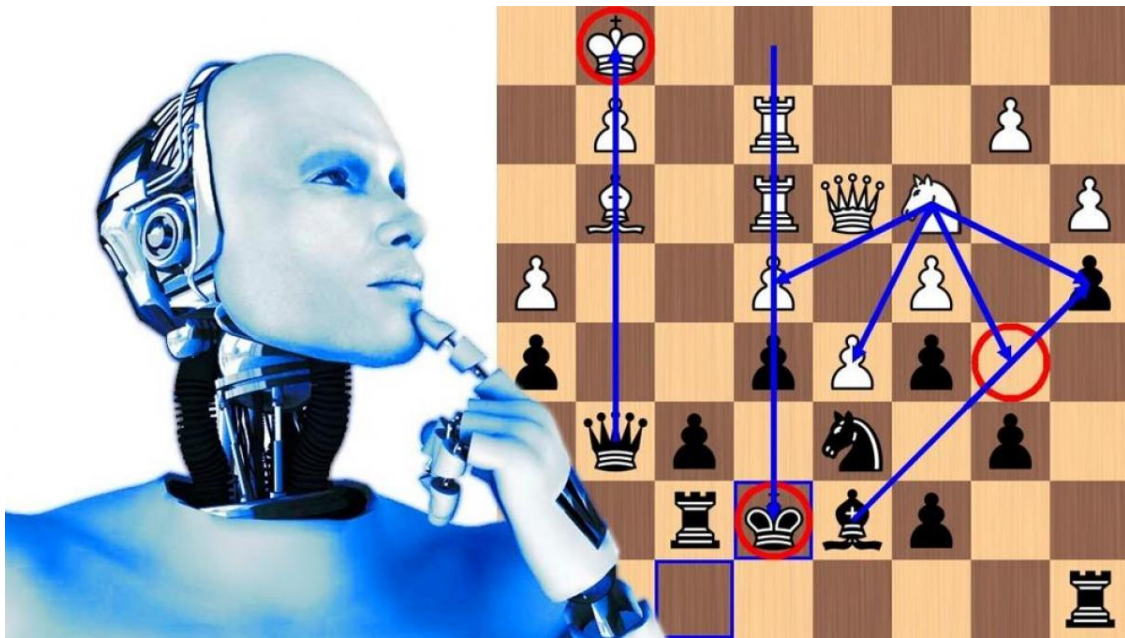


Universitat de Barcelona

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

<b>1</b>	<b>Introducció</b>	<b>2</b>
<b>2</b>	<b>Explicacions</b>	<b>3</b>
2.1	Funció <code>is_checkmate()</code> i taules . . . . .	3
2.2	Implementació de l'algorisme Minimax . . . . .	3
2.3	Implementació de l'algorisme Minimax amb poda $\alpha\text{-}\beta$ . . . . .	4
2.4	Implementació de l'algorisme Expectimax . . . . .	4
<b>3</b>	<b>Respostes</b>	<b>6</b>
3.1	Blanques: Minimax (profunditat 4) Negres: Minimax (profunditat 4) . . . . .	6
3.2	Blanques: Minimax (profunditat 1-4) Negres: Minimax (profunditat 1-4) . . . . .	7
3.3	Blanques: Minimax (profunditat 4) Negres: Poda $\alpha\text{-}\beta$ (profunditat 4) . . . . .	7
3.4	Blanques: Poda $\alpha\text{-}\beta$ (profunditat 1-5) Negres: Poda $\alpha\text{-}\beta$ (profunditat 1-5) . . . . .	8
3.5	Blanques: Expectimax (profunditat 1-4) Negres: Poda $\alpha\text{-}\beta$ (profunditat 1-4) . . . . .	9
3.6	Preguntes finals . . . . .	9
<b>4</b>	<b>Conclusions i valoracions personals</b>	<b>11</b>
<b>A</b>	<b>Codi de les funció <code>is_checkmate()</code> comentat</b>	<b>12</b>
<b>B</b>	<b>Codis de la implementació de l'algorisme Minimax comentats</b>	<b>13</b>
<b>C</b>	<b>Codis de la implementació de l'algorisme Minimax amb poda <math>\alpha\text{-}\beta</math> comentats</b>	<b>16</b>
<b>D</b>	<b>Codis de la implementació de l'algorisme Expectimax comentats</b>	<b>20</b>

# 1 Introducció

Els jocs amb oponents són una peça clau en el desenvolupament de la Intel·ligència Artificial. Aquests algorismes 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  $\alpha$ - $\beta$  i expectimax (juntament amb diverses altres funcions necessàries) per simular una partida d'escacs amb un rei blanc, un rei negre, una torre blanca i una torre negra. També analitzarem, variant i convinant els anteriors algorismes entre si, les sortides per pantalla que obtindrem.

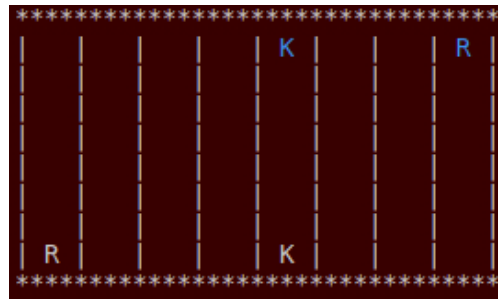


Figure 1: Representació de l'estat inicial de les peces.

## 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 inicialment 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, "defensors". 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 defensat 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 defensat. 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 defensat no s'interposi en l'escac i mat, si es que encara està a la partida. El cas de les files serà 100% anàlog.

Explicuem 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 produït 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

$(move, value) \leftarrow \text{MAX-VALUE}(u, player)$

**return**  $move$

**function** MAX-VALUE( $u, player$ ) :

**if**  $terminal(u)$  **then**

**return**  $(NULL, utility(u, player))$

**end**

$(best-move, max-value) \leftarrow (NULL, -\infty)$

**for**  $max-move \in moves(u)$  **do**

$(min-move, value) \leftarrow \text{MIN-VALUE}(result(u, move), player)$

**if**  $value > max-value$  **then**

$(best-move, max-value) \leftarrow (max-move, value)$

**end**

**end**

**return**  $(best-move, max-value)$

**end**

**function** MIN-VALUE( $u, player$ ) :

**if**  $terminal(u)$  **then**

**return**  $(NULL, utility(u, player))$

**end**

$(best-move, min-value) \leftarrow (NULL, +\infty)$

**for**  $min-move \in moves(u)$  **do**

$(max-move, value) \leftarrow \text{MAX-VALUE}(result(u, move), player)$

**if**  $value < min-value$  **then**

$(best-move, min-value) \leftarrow (max-move, value)$

**end**

**end**

**return**  $(best-move, min-value)$

**end**

---

Figure 2: Pseudocodi de l'algorisme Minimax

### 2.3 Implementació de l'algorisme Minimax amb poda $\alpha$ - $\beta$

La implementació del minimax amb la poda  $\alpha$ - $\beta$  és exactament igual que amb el minimax però afegint els paràmetres  $\alpha$  i  $\beta$  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  $\alpha$ - $\beta$  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,  $\alpha$ ,  $\beta$ , jugador)
  si nodo es un nodo terminal o profundidad = 0
    devolver el valor heurístico del nodo
  si jugador1
    para cada hijo de nodo
       $\alpha := \max(\alpha, \text{alfa-beta}(\text{hijo}, \text{profundidad}-1, \alpha, \beta, \text{jugador2}))$ 
      si  $\beta \leq \alpha$ 
        romper (* poda  $\beta$  *)
    devolver  $\alpha$ 
  si no
    para cada hijo de nodo
       $\beta := \min(\beta, \text{alfa-beta}(\text{hijo}, \text{profundidad}-1, \alpha, \beta, \text{jugador1}))$ 
      si  $\beta \leq \alpha$ 
        romper (* poda  $\alpha$  *)
    devolver  $\beta$ 

```

Figure 3: Pseudocodi de l'algorisme Poda  $\alpha$ - $\beta$

---

**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

$(move, value) \leftarrow \text{EXPECTIMAX-VALUE}(u, player)$

**return**  $move$

**function** EXPECTIMAX-VALUE( $u, player$ ) :

**if**  $u$  is a chance node **then**

$expected \leftarrow 0$

**for**  $r \in \text{moves}(u)$  **do**

$(move, value) \leftarrow \text{EXPECTIMAX-VALUE}(\text{result}(u, r),$   
        $player)$

$expected \leftarrow expected + P(r) \cdot value$

**end**

**return**  $(NULL, expected)$

**else if**  $\text{turn}(u) = player$  **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 deduir que així serà també si seguïssim 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, les 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 negra i impedit 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:

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

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 $\alpha$ - $\beta$ (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 diferència 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  $\alpha$ - $\beta$  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  $\alpha$ - $\beta$  cal entendre com funciona l'algorisme amb poda  $\alpha$ - $\beta$ . 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  $\alpha$ - $\beta$  a l'algorisme minimax.

## 3.4 Blanques: Poda $\alpha$ - $\beta$ (profunditat 1-5) Negres: Poda $\alpha$ - $\beta$ (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  $\alpha$ - $\beta$  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  $\alpha$ - $\beta$  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 $\alpha$ - $\beta$ (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:

<div>BLACK WHITE</div>	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 perquè 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 altra 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.

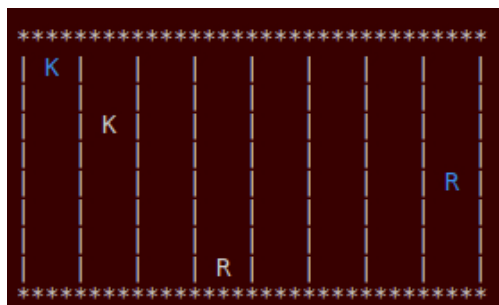


Figure 10: Exemple de taulell on les blanques guanyarien el 100% de les vegades amb un sol moviment.

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. Inicialment, 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 decidir 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

```
1 def is_Checkmate(self,currentState,color):
2     #Per no duplicar el codi, ho farem sense tenir en compte el color, nomes a la
3     #inicialitzacio
4     #A = el que li toca jugar, "ataca"    D = el que no li toca jugar, "defensa"
5
6     #Inicialitzem totes les variables
7     if color:
8         dkState = self.getPieceState(currentState, 12)
9         akState = self.getPieceState(currentState, 6)
10        arState = self.getPieceState(currentState, 2)
11        drState = self.getPieceState(currentState, 8)
12    else:
13        akState = self.getPieceState(currentState, 12)
14        dkState = self.getPieceState(currentState, 6)
15        drState = self.getPieceState(currentState, 2)
16        arState = self.getPieceState(currentState, 8)
17
18    filaAk = akState[0]
19    columnaAk = akState[1]
20    filaDk = dkState[0]
21    columnaDk = dkState[1]
22
23    if arState != None:
24        filaAr = arState[0]
25        columnaAr = arState[1]
26    else:
27        return False #No es possible conseguir checkMate amb nomes el rei
28
29    if drState != None:
30        filaDr = drState[0]
31        columnaDr = drState[1]
32
33    #Nomes estarem en situacio d'escac i mat si el rei D esta a les files o columnes 0 o 7
34    #Mirem inicialment les columnes
35    if (columnaDk == 0) or (columnaDk == 7):
36        #La torre A ha d'estar a la mateixa, pero no el costat del rei D
37        if (columnaDk == columnaAr) and (abs(filaDk - filaAr) > 1):
38            #El rei A ha d'estar a dues columnes de separacio i a la mateixa fila o una mes,
39            #si es troba a una cantonada
40            if abs(columnaDk - columnaAk) == 2 and ((filaDk == filaAk) or (abs(filaDk -
41            filaAk) == 1 and (filaDk == 0 or filaDk == 7))):
42                if drState == None:
43                    return True
44                if(filaAk != filaDr and columnaAk != columnaDr):
45                    if(filaAk == filaDk):
46                        if(filaAr < filaDk and (filaDr < filaAr or filaDr > filaDk)) or
47                        (filaAr > filaDk and (filaDr > filaAr or filaDr < filaDk)):
48                            return True
49                    #Mirem aqui les quatre configuracions diferents (rei que es defensa a
50                    #una cantonada i al que ataca a una fila diferent)
51                    if(filaAr < filaDk and (filaDr < filaAr or filaDr == filaDk)) or
52                    (filaAr > filaDk and (filaDr > filaAr or filaDr == filaDk)):
53                        return True
54
55    #De manera analoga mirarem les files
56    if (filaDk == 0) or (filaDk == 7):
57        if (filaDk == filaAr and abs(columnaDk - columnaAr) > 1) or ((abs(columnaDk -
58        columnaAk) == 1 and (columnaDk == 0 or columnaDk == 7)):
59            if (abs(filaDk - filaAk) == 2 and columnaDk == columnaAk):
60                if drState == None:
```

```

61         return True
62     if(columnaAk == columnaDk)
63         if(filaAk != filaDr and columnaAk != columnaDr):
64             if(columnaAr < columnaDk and (columnaDr < columnaAr or columnaDr
65             > columnaDk)) or (columnaAr > columnaDk and (columnaDr > columnaAr
66             or columnaDr < columnaDk)):
67                 return True
68             if(columnaAr < columnaDk and (columnaDr < columnaAr or columnaDr ==
69             columnaDk)) or (columnaAr > columnaDk and (columnaDr > columnaAr or
70             columnaDr == columnaDk)):
71                 return True
72
73 #Si no estem en cap situacio anterior, retornem false
74 return False

```

## B Codis de la implementació de l'algorisme Minimax comentats

```

1 def minimaxGame(self, depthWhite, depthBlack):
2     currentState = self.getCurrentState()
3     currentStateTuple = tuple(tuple(row) for row in currentState)
4     color = True #Primer mouen les blanques
5     movSenseMenjar = 0 #Si arriba a 50, son taules
6     nombre_fitxes = len(currentState)
7     ciclic = False
8     self.dictVisitedStates = {currentStateTuple : 1} #Si el valor d'alguna clau arriba a 3,
9     #son taules
10    self.pathToTarget.append(copy.deepcopy(currentState))
11    self.minimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())
12
13    #A cada torn anem cridant la funcio minimaxDecision i mirem si es escac i mat o taules
14    while (not self.is_Checkmate(self.getCurrentState(), color)) and
15    len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:
16        currentState = self.getCurrentState()
17        currentStateTuple = tuple(tuple(row) for row in currentState)
18
19        self.pathToTarget.append(copy.deepcopy(currentState))
20
21        #Analitzem si s'ha menjat una peca
22        if(len(currentState) < nombre_fitxes):
23            nombre_fitxes = len(currentState)
24            movSenseMenjar = 0
25        else:
26            movSenseMenjar += 1
27
28        #Actualitzem el diccionari d'estats visitats
29        if currentStateTuple in self.dictVisitedStates.keys():
30            self.dictVisitedStates[currentStateTuple] += 1
31            if self.dictVisitedStates[currentStateTuple] == 3:
32                ciclic = True
33        else:
34            self.dictVisitedStates[currentStateTuple] = 1
35
36        color = not color
37        aichess.chess.board.print_board()
38        self.minimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())
39
40    currentState = self.getCurrentState()
41    self.pathToTarget.append(copy.deepcopy(currentState))
42
43    #Mostrem per pantalla el resultat

```

```

44     if self.is_Checkmate(currentState, color):
45         if color:
46             print("Guanyen les blanques!")
47         else:
48             print("Guanyen les negres!")
49     else:
50         print("Taules!")
51
52     #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
53     aichess.chess.board.print_board()
54
55
56 def minimaxDecision(self, depthWhite, depthBlack, color, currentState):
57     if(color):
58         depth = depthWhite
59     else:
60         depth = depthBlack
61
62     self.max_value(0,color,currentState,depth)
63
64
65 def max_value(self, currentDepth, color, currentState, depthMax):
66     if(currentDepth == depthMax): #Estat terminal
67         return self.heuristica(currentState, color)
68
69     v=float("-inf")
70
71     #Creem un nou tauler amb l'estat actual
72     self.newBoardSim(currentState)
73
74     if(color):
75         #Mirem tots els següents estats possibles
76         for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
77
78             #Mirem el moviment que es produeix i el fem
79             movement = self.getMovement(self.getWhiteState(currentState),nextState)
80             aichess.chess.moveSim(movement[0],movement[1],False)
81
82             #Comprovem si l'estat actual es checkmate
83             if(self.is_Checkmate(self.getCurrentSimState(), color)):
84                 if(currentDepth == 0):
85                     nextMove = movement
86                     aichess.chess.move(nextMove[0],nextMove[1])
87                     self.newBoardSim(self.getCurrentState())
88                     return self.heuristica(self.getCurrentSimState(), color)
89
90             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
91             #les negres
92             if not(self.isWatchedWk(self.getCurrentSimState())):
93                 s = max(v,self.min_value(currentDepth+1,not color,
94                     self.getCurrentSimState(),depthMax))
95                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
96                 if(currentDepth == 0):
97                     if(v < s):
98                         nextMove = movement
99                     v = s
100
101             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
102             self.newBoardSim(currentState)
103
104     else:
105         #Mirem tots els següents estats possibles

```

```

106         for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
107             #Mirem el moviment que es produeix i el fem
108             movement = self.getMovement(self.getBlackState(currentState),nextState)
109             aichess.chess.moveSim(movement[0],movement[1],False)
110
111             #Comprovem si l'estat actual es checkmate
112             if(self.is_Checkmate(self.getCurrentSimState(), color)):
113                 if(currentDepth == 0):
114                     nextMove = movement
115                     aichess.chess.move(nextMove[0],nextMove[1])
116                     self.newBoardSim(self.getCurrentState())
117                     return self.heuristica(self.getCurrentSimState(), color)
118
119             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
120             #les blanques
121             if not(self.isWatchedBk(self.getCurrentSimState())):
122                 s = max(v,self.min_value(currentDepth+1,not color,
123                 self.getCurrentSimState(),depthMax))
124                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
125                 if(currentDepth == 0):
126                     if(v < s):
127                         nextMove = movement
128                 v = s
129
130             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
131             self.newBoardSim(currentState)
132
133         #Una vegada trobat el millor moviment, el fem al tauler real
134         if(currentDepth == 0):
135             aichess.chess.move(nextMove[0],nextMove[1])
136             self.newBoardSim(self.getCurrentState())
137
138         return v
139
140
141 def min_value(self, currentDepth, color, currentState, depthMax):
142     if(currentDepth == depthMax): #Estat terminal
143         return self.heuristica(currentState, not color)
144
145     v=float("inf")
146
147     #Creem un nou tauler amb l'estat actual
148     self.newBoardSim(currentState)
149
150     if(color):
151         #Mirem tots els següents estats possibles
152         for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
153
154             #Mirem el moviment que es produeix i el fem
155             movement = self.getMovement(self.getWhiteState(currentState),nextState)
156             aichess.chess.moveSim(movement[0],movement[1],False)
157
158             #Comprovem si l'estat actual es checkmate
159             if(self.is_Checkmate(self.getCurrentSimState(), color)):
160                 return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal
161
162             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
163             #les negres
164             if not(self.isWatchedWk(self.getCurrentSimState())):
165                 s = min(v,self.max_value(currentDepth+1,not color,
166                 self.getCurrentSimState(),depthMax))
167                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica

```



```

168         if(currentDepth == 0):
169             if(v > s):
170                 nextMove = movement
171             v = s
172
173         #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
174         self.newBoardSim(currentState)
175
176     else:
177         #Mirem tots els següents estats possibles
178         for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
179
180             #Mirem el moviment que es produeix i el fem
181             movement = self.getMovement(self.getBlackState(currentState),nextState)
182             aichess.chess.moveSim(movement[0],movement[1],False)
183
184             #Comprovem si l'estat actual es checkmate
185             if(self.is_Checkmate(self.getCurrentSimState(), color)):
186                 return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal
187
188             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
189             #les blanques
190             if not(self.isWatchedBk(self.getCurrentSimState())):
191                 s = min(v,self.max_value(currentDepth+1,not color,
192                 self.getCurrentSimState(),depthMax))
193                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
194                 if(currentDepth == 0):
195                     if(v > s):
196                         nextMove = movement
197                     v = s
198
199             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
200             self.newBoardSim(currentState)
201
202         #Si això passa, vol dir que no es pot fer cap moviment legal i, per tant, el moviment
203         #es dolent
204         if(v==float('inf')):
205             v = float('-inf')
206
207     return v

```

## C Codis de la implementació de l'algorisme Minimax amb poda $\alpha$ - $\beta$ comentats

```

1 def alphaBetaPodaGame(self, depthWhite,depthBlack):
2
3     currentState = self.getCurrentState()
4     currentStateTuple = tuple(tuple(row) for row in currentState)
5     color = True; #Primer mouen les blanques
6     movSenseMenjar = 0
7     nombre_fitxes = len(currentState)
8     ciclic = False
9     self.dictVisitedStates = {currentStateTuple : 1}
10    self.pathToTarget.append(copy.deepcopy(currentState))
11    self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())
12
13    #A cada torn anem cridant la funcio alphaBetaPodaDecision i mirem si es escac i mat
14    #o taules
15    while (not self.is_Checkmate(self.getCurrentState(), color)) and
16    len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:

```

```

17     currentState = self.getCurrentState()
18     currentStateTuple = tuple(tuple(row) for row in currentState)
19     self.pathToTarget.append(copy.deepcopy(currentState))
20
21     #Analitzem si s'ha menjat una peca
22     if(len(currentState) < nombre_fitxes):
23         nombre_fitxes = len(currentState)
24         movSenseMenjar = 0
25     else:
26         movSenseMenjar += 1
27
28     #Actualitzem el diccionari d'estats visitats
29     if currentStateTuple in self.dictVisitedStates.keys():
30         self.dictVisitedStates[currentStateTuple] += 1
31         if self.dictVisitedStates[currentStateTuple] == 3:
32             ciclic = True
33     else:
34         self.dictVisitedStates[currentStateTuple] = 1
35
36     color = not color
37     aichess.chess.board.print_board()
38     self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())
39
40     self.pathToTarget.append(copy.deepcopy(currentState))
41
42     #Mostrem per pantalla el resultat
43     if self.is_Checkmate(self.getCurrentState(), color):
44         if color:
45             print("Guanyen les blanques!")
46         else:
47             print("Guanyen les negres!")
48     else:
49         print("Taules!")
50
51     #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
52     aichess.chess.board.print_board()
53
54
55 def alphaBetaPodaDecision(self, depthWhite, depthBlack, color, currentState):
56     if(color):
57         depth = depthWhite
58     else:
59         depth = depthBlack
60
61     self.alphaBetaMax(0,color,currentState,depth,float('-inf'),float('inf'))
62
63
64 def alphaBetaMax(self, currentDepth, color, currentState, depthMax, alpha, beta):
65     if(currentDepth == depthMax): #Estat terminal
66         return self.heuristica(currentState, color)
67
68     v=float("-inf")
69
70     #Creem un nou tauler amb l'estat actual
71     self.newBoardSim(currentState)
72
73     if(color):
74         #Mirem tots els següents estats possibles
75         for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
76
77             #Mirem el moviment que es produeix i el fem
78             movement = self.getMovement(self.getWhiteState(currentState),nextState)

```

```

79         aichess.chess.moveSim(movement[0],movement[1],False)
80
81     #Comprovem si l'estat actual es checkmate
82     if(self.is_Checkmate(self.getCurrentSimState(), color)):
83         if(currentDepth == 0):
84             nextMove = movement
85             aichess.chess.move(nextMove[0],nextMove[1])
86             self.newBoardSim(self.getCurrentState())
87             return self.heuristica(self.getCurrentSimState(), color)
88
89     #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
90     #les negres
91     if not(self.isWatchedWk(self.getCurrentSimState())):
92         s = max(v,self.alphaBetaMin(currentDepth+1,not color,
93             self.getCurrentSimState(),depthMax,alpha,beta))
94         #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
95         if(currentDepth == 0):
96             if(v < s):
97                 nextMove = movement
98             v = s
99             if v >= beta:
100                 return v
101             alpha = max(alpha, v)
102
103     #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
104     self.newBoardSim(currentState)
105
106 else:
107     #Mirem tots els següents estats possibles
108     for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
109         #Mirem el moviment que es produeix i el fem
110         movement = self.getMovement(self.getBlackState(currentState),nextState)
111         aichess.chess.moveSim(movement[0],movement[1],False)
112
113     #Comprovem si l'estat actual es checkmate
114     if(self.is_Checkmate(self.getCurrentSimState(), color)):
115         if(currentDepth == 0):
116             nextMove = movement
117             aichess.chess.move(nextMove[0],nextMove[1])
118             self.newBoardSim(self.getCurrentState())
119             return self.heuristica(self.getCurrentSimState(), color)
120
121     #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
122     #les blanques
123     if not(self.isWatchedBk(self.getCurrentSimState())):
124         s = max(v,self.alphaBetaMin(currentDepth+1,not color,
125             self.getCurrentSimState(),depthMax,alpha,beta))
126         #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
127         if(currentDepth == 0):
128             if(v < s):
129                 nextMove = movement
130             v = s
131             if v >= beta:
132                 return v
133             alpha = max(alpha, v)
134
135     #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
136     self.newBoardSim(currentState)
137
138     #Una vegada trobat el millor moviment, el fem al tauler real
139     if(currentDepth == 0):
140         aichess.chess.move(nextMove[0],nextMove[1])

```

```

141         self.newBoardSim(self.getCurrentState())
142
143     return v
144
145
146 def alphaBetaMin(self, currentDepth, color, currentState, depthMax, alpha, beta):
147     if(currentDepth == depthMax): #Estat terminal
148         return self.heuristica(currentState, not color)
149
150     v=float("inf")
151
152     #Creem un nou tauler amb l'estat actual
153     self.newBoardSim(currentState)
154
155     if(color):
156         #Mirem tots els següents estats possibles
157         for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
158
159             #Mirem el moviment que es produeix i el fem
160             movement = self.getMovement(self.getWhiteState(currentState),nextState)
161             aichess.chess.moveSim(movement[0],movement[1],False)
162
163             #Comprovem si l'estat actual es checkmate
164             if(self.is_Checkmate(self.getCurrentSimState(), color)):
165                 return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal
166
167             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
168             #les negres
169             if not(self.isWatchedWk(self.getCurrentSimState())):
170                 s = min(v,self.alphaBetaMax(currentDepth+1,not color,
171                 self.getCurrentSimState(),depthMax,alpha,beta))
172                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica
173                 if(currentDepth == 0):
174                     if(v > s):
175                         nextMove = movement
176                 v = s
177                 if v <= alpha:
178                     return v
179                 beta = min(beta, v)
180
181             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
182             self.newBoardSim(currentState)
183
184     else:
185
186         #Mirem tots els següents estats possibles
187         for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
188
189             #Mirem el moviment que es produeix i el fem
190             movement = self.getMovement(self.getBlackState(currentState),nextState)
191             aichess.chess.moveSim(movement[0],movement[1],False)
192
193             #Comprovem si l'estat actual es checkmate
194             if(self.is_Checkmate(self.getCurrentSimState(), color)):
195                 return self.heuristica(self.getCurrentSimState(), not color) #Estat terminal
196
197             #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check de
198             #les blanques
199             if not(self.isWatchedBk(self.getCurrentSimState())):
200                 s = min(v,self.alphaBetaMax(currentDepth+1,not color,
201                 self.getCurrentSimState(),depthMax,alpha,beta))
202                 #Si estem a l'estat inicial, guardem el moviment amb millor heuristica

```

```

203         if(currentDepth == 0):
204             if(v > s):
205                 nextMove = movement
206             v = s
207             if v <= alpha:
208                 return v
209             beta = min(beta, v)
210
211             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
212             self.newBoardSim(currentState)
213
214     if(currentDepth == 0):
215         aichess.chess.move(nextMove[0],nextMove[1])
216         self.newBoardSim(self.getCurrentState())
217
218     #Si això passa, vol dir que no es pot fer cap moviment legal i, per tant, el moviment
219     #es dolent
220     if(v==float('inf')):
221         v = float('-inf')
222
223     return v

```

## D Codis de la implementació de l'algorisme Expectimax comentats

```

1 def expectimaxGame(self, depthWhite, depthBlack):
2     currentState = self.getCurrentState()
3     currentStateTuple = tuple(tuple(row) for row in currentState)
4     color = True; #Primer mouen les blanques
5     movSenseMenjar = 0
6     nombre_fitxes = len(currentState)
7     ciclic = False
8     self.dictVisitedStates = {currentStateTuple : 1}
9     self.pathToTarget.append(copy.deepcopy(currentState))
10    self.expectimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())
11
12    #A cada torn anem cridant la funció minimaxDecision i mirem si es escac i mat o taules
13    while (not self.is_Checkmate(self.getCurrentState(), color))
14    and len(self.getCurrentState()) > 2 and (movSenseMenjar <= 50) and not ciclic:
15        currentState = self.getCurrentState()
16        currentStateTuple = tuple(tuple(row) for row in currentState)
17        self.pathToTarget.append(copy.deepcopy(currentState))
18
19        #Analitzem si s'ha menjat una peça
20        if(len(currentState) < nombre_fitxes):
21            nombre_fitxes = len(currentState)
22            movSenseMenjar = 0
23        else:
24            movSenseMenjar += 1
25
26        #Actualitzem el diccionari d'estats visitats
27        if currentStateTuple in self.dictVisitedStates.keys():
28            self.dictVisitedStates[currentStateTuple] += 1
29            if self.dictVisitedStates[currentStateTuple] == 3:
30                ciclic = True
31        else:
32            self.dictVisitedStates[currentStateTuple] = 1
33
34        color = not color
35        aichess.chess.board.print_board()
36

```

```

37         if(color):
38             self.expectimaxDecision(depthWhite, depthBlack, color, self.getCurrentState())
39         else:
40             self.alphaBetaPodaDecision(depthWhite, depthBlack, color, self.getCurrentState())
41
42     currentState = self.getCurrentState()
43     self.pathToTarget.append(copy.deepcopy(currentState))
44
45     #Mostrem per pantalla el resultat
46     if self.is_Checkmate(currentState, color):
47         if color:
48             print("Guanyen les blanques!")
49         else:
50             print("Guanyen les negres!")
51     else:
52         print("Taules!")
53
54     #Mostrem per pantalla el taulell final, aixi com l'hem anat mostrant cada torn
55     aichess.chess.board.print_board()
56
57 def expectimaxDecision(self, depthWhite, depthBlack, color, currentState):
58     if(color):
59         depth = depthWhite
60     else:
61         depth = depthBlack
62
63     self.expectimax(0,color,currentState,depth,"max","None")
64
65 def expectimax(self, currentDepth, color, currentState, depthMax, nodeType,
66 previousNodeType):
67     nextMove = []
68     values = []
69     if(currentDepth == depthMax): #Estat terminal
70         if(nodeType == "min" or previousNodeType == "max"):
71             return self.heuristica(currentState, not color)
72         return self.heuristica(currentState, color)
73
74     if(nodeType == "max"):
75         v=float('-inf')
76     else:
77         v=float('inf')
78
79     #Creem un nou tauler amb l'estat actual
80     self.newBoardSim(currentState)
81
82     if(color):
83
84         #Mirem tots els següents estats possibles
85         for nextState in self.getListNextStatesW(self.getWhiteState(currentState)):
86             #Si estem a un node chance, calculem el valor de cada estat successor i l'afegim
87             #a la llista values
88             if(nodeType == "chance"):
89                 #Si el node anterior era max, el següent sera min i viceversa
90                 if(previousNodeType == "max"):
91                     values.append(self.expectimax(currentDepth+1, color,
92 self.getCurrentSimState(), depthMax, nodeType="min",
93 previousNodeType="chance"))
94                 elif(previousNodeType == "min"):
95                     values.append(self.expectimax(currentDepth+1, color,
96 self.getCurrentSimState(), depthMax, nodeType="max",
97 previousNodeType="chance"))
98

```

```

99         #Si estem a un node min o max
100     else:
101         #Mirem el moviment que es produeix i el fem
102         movement = self.getMovement(self.getWhiteState(currentState), nextState)
103         aichess.chess.moveSim(movement[0], movement[1], False)
104
105         #Comprovem si l'estat actual es checkmate
106         if(self.is_Checkmate(self.getCurrentSimState(), color)):
107             if(currentDepth == 0):
108                 nextMove = movement
109                 aichess.chess.move(nextMove[0], nextMove[1])
110                 self.newBoardSim(self.getCurrentState())
111                 return self.heuristica(self.getCurrentSimState(), color)
112
113         #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check
114         #de les negres
115         if not(self.isWatchedWk(self.getCurrentSimState())):
116             #Calculem el maxim o minim dels nodes chance successors
117             if(nodeType == "max"):
118                 s = max(v, self.expectimax(currentDepth+1, not color,
119                 self.getCurrentSimState(), depthMax, nodeType="chance",
120                 previousNodeType="max"))
121             elif(nodeType == "min"):
122                 s = min(v, self.expectimax(currentDepth+1, not color,
123                 self.getCurrentSimState(), depthMax, nodeType="chance",
124                 previousNodeType="min"))
125
126             #Si estem a l'estat inicial, guardem el moviment amb millor esperanca
127             if(currentDepth == 0):
128                 if(v < s):
129                     nextMove = movement
130                 v = s
131
132             #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
133             self.newBoardSim(currentState)
134
135     else:
136         #Mirem tots els següents estats possibles
137         for nextState in self.getListNextStatesB(self.getBlackState(currentState)):
138             #Si estem a un node chance, calculem el valor de cada estat successor i l'afegim
139             #a la llista values
140             if(nodeType == "chance"):
141                 #Si el node anterior era max, el següent sera min i viceversa
142                 if(previousNodeType == "max"):
143                     values.append(self.expectimax(currentDepth+1, color,
144                     self.getCurrentSimState(), depthMax, nodeType="min",
145                     previousNodeType="chance"))
146                 elif(previousNodeType == "min"):
147                     values.append(self.expectimax(currentDepth+1, color,
148                     self.getCurrentSimState(), depthMax, nodeType="max",
149                     previousNodeType="chance"))
150             #Si estem a un node min o max
151         else:
152             #Mirem el moviment que es produeix i el fem
153             movement = self.getMovement(self.getBlackState(currentState), nextState)
154             aichess.chess.moveSim(movement[0], movement[1], False)
155
156             #Comprovem si l'estat actual es checkmate
157
158             if(self.is_Checkmate(self.getCurrentSimState(), color)):
159                 if(currentDepth == 0):
160                     nextMove = movement

```

```

161         aichess.chess.move(nextMove[0],nextMove[1])
162         self.newBoardSim(self.getCurrentState())
163         return self.heuristica(self.getCurrentSimState(), color)
164
165     #Comprovem que el moviment sigui legal, es a dir, que no es fiqui en check
166     #de les blanques
167     if not(self.isWatchedBk(self.getCurrentSimState())):
168         #Calculem el maxim o minim dels nodes chance successors
169         if(nodeType == "max"):
170             s = max(v,self.expectimax(currentDepth+1, not color,
171             self.getCurrentSimState(), depthMax, nodeType="chance",
172             previousNodeType="max"))
173         elif(nodeType == "min"):
174             s = min(v, self.expectimax(currentDepth+1, not color,
175             self.getCurrentSimState(), depthMax, nodeType="chance",
176             previousNodeType="min"))
177
178         #Si estem a l'estat inicial, guardem el moviment amb millor esperanca
179         if(currentDepth == 0):
180             if(v < s):
181                 nextMove = movement
182             v = s
183
184         #Tornem a crear un nou tauler amb l'estat inicial, es a dir, desfem el moviment
185         self.newBoardSim(currentState)
186
187     #Una vegada coneguts tots els valors dels estats successors calculem l'esperanca del
188     #node chance
189     if(nodeType == "chance"):
190         v = self.calculateValue(values)
191
192     if(currentDepth == 0):
193         aichess.chess.move(nextMove[0],nextMove[1])
194         self.newBoardSim(self.getCurrentState())
195
196     return v

```