
# ANNEX: APRENENTATGE AUTOMÀTIC APLICAT AL BLACKJACK

Pau Amorós Faro
2n Batxillerat B
Bernat Vidal
Treball de Recerca II

# ÍNDEX

**0. NOTA DE L'AUTOR** **2**
**1. RL BLACKJACK - PROGRAMA INICIAL** **3**
1.1 RESULTATS 4
**2. MONTE CARLO - PROGRAMA DEFINITIU I COMPLET** **12**
2.1 RESULTATS 23
**3. GLOSSARI DELS MÈTODES UTILITZATS** **24**

# 0. NOTA DE L’AUTOR

La versió del programa ha estat realitzada completament dia 16/04/23. En conseqüència, qualsevol canvi en la interfície de les llibreries usades o funcions exportades en un futur podria incapacitar i/o anul·lar el programa completament.

A més, si es pretén executar el codi que es mostrarà a continuació des d’un dispositiu propi es recomana eliminar els comentaris afegits de llargària significativa, ja que poden arribar a causar interferències.

# 1. RL BLACKJACK - PROGRAMA INICIAL

#Codi d'en Pau Amorós Faro IES Alcúdia ---- TR: Aprenetatge automàtic aplicat al Blackjack

```
import rlcard      #Importem dos mòduls del "paquet" rlcard.
from rlcard.agents import DQNAgent      #Aquí importem una classe que implementa un algorisme Deep-Q.

entorn = rlcard.make("blackjack") #Creem una instància de l'entorn del blackjack utilitzant el mètode make().

agent = DQNAgent(          #Creem una instància de la classe DQNAgent
    num_actions=entorn.num_actions,     #Nombre d'accions que es poden prendre
    state_shape=entorn.state_shape[0],    #Forma de l'estat representat (vector unidimensional)
    mlp_layers=[64,64],       #Nombre de neurons a cada capa de la xarxa neuronal.
)

entorn.set_agents([agent]) #L'agent creat abans és l'únic a l'entorn.

from rlcard.utils import (
    tournament,        #Manera en la qual l'agent s'entrenarà.
    reorganize,        #Reorganitzar les dades d'una manera òptima per elaborar gràfics.
    Logger,     #Una classe per acumular informació durant l'entrenament.
    plot_curve,        #Una funció per determinar la corva d'aprenentatge.
)

with Logger("experiments/leduc_holdem_dqn_result/") as logger:
    for episode in range(1000):      #El bucle d'entrenament serà de 1000 episodis.
```

```
        trajectories, payoffs = entorn.run(is_training=True)        #Per a cada episodi, les trajectòries i pagaments de l'entorn es graven.

        trajectories = reorganize(trajectories, payoffs)      #Les trajectòries es reorganitzen per a la màxima eficiència durant l'entrenament i l'agent reb els resultats.

        for ts in trajectories[0]:
            agent.feed(ts)

        if episode % 50 == 0:           #Per a cada 50 episodis, el rendiment del agent es guarda a la classe Logger, la qual ho emmagatzema a un arxiu CSV i genera un gràfic del rendiment.
            logger.log_performance(
                entorn.timestep,
                tournament(
                    entorn,
                    10000,
                )[0]
            )

    csv_path, fig_path = logger.csv_path, logger.fig_path        #El lloc en el qual s'envien les dades.

plot_curve(csv_path, fig_path, "DQN")    #Es representa gràficament l'evolució de l'estil de joc de l'agent.
```

## 1.1 RESULTATS

A continuació es mostren els resultats exactes en valor numèric. Aquests mateixos s'han simplificat en un gràfic a la memòria tècnica del treball per a millorar la comprensió i per motius pràctics.

```
--------------------------------------
 episode      | 1
 reward       | -0.9247
--------------------------------------


--------------------------------------
 episode      | 18780
 reward       | -0.9191
--------------------------------------
INFO - Step 100, rl-loss: 1.036237120628357
INFO - Copied model parameters to target network.
INFO - Step 132, rl-loss: 0.64948683977127087
--------------------------------------
 episode      | 37650
 reward       | -0.7887
--------------------------------------
INFO - Step 205, rl-loss: 0.45411744713783264
--------------------------------------
 episode      | 56394
 reward       | -0.0914
--------------------------------------
INFO - Step 274, rl-loss: 0.39138972759246826
--------------------------------------
 episode      | 71021
 reward       | -0.1313
--------------------------------------
INFO - Step 346, rl-loss: 0.34966084361076355
--------------------------------------
 episode      | 82860
```

| reward | -0.1866 |
|---|---|

---

INFO - Step 418, rl-loss: 0.48397752642631536

---

| episode | 93211 |
|---|---|
| reward | -0.176 |

---

INFO - Step 487, rl-loss: 0.45444995164871216

---

| episode | 103813 |
|---|---|
| reward | -0.177 |

---

INFO - Step 554, rl-loss: 0.29446095228195195

---

| episode | 114506 |
|---|---|
| reward | -0.1823 |

---

INFO - Step 625, rl-loss: 0.51690632104873667

---

| episode | 125152 |
|---|---|
| reward | -0.1821 |

---

INFO - Step 686, rl-loss: 0.64734488725662236

---

| episode | 135858 |
|---|---|
| reward | -0.1845 |

---

INFO - Step 754, rl-loss: 0.32832098007202153

---

| episode | 146622 |
|---|---|
| reward | -0.1916 |

```
---------------------------------------
INFO - Step 825, rl-loss: 0.29668548703193665
---------------------------------------
  episode      |  156945
  reward       |  -0.1936
---------------------------------------
INFO - Step 893, rl-loss: 0.58128017187118537
---------------------------------------
  episode      |  167222
  reward       |  -0.1776
---------------------------------------
INFO - Step 966, rl-loss: 0.38330298662185676
---------------------------------------
  episode      |  177721
  reward       |  -0.1815
---------------------------------------
INFO - Step 1033, rl-loss: 0.31779187917709355
---------------------------------------
  episode      |  188026
  reward       |  -0.1787
---------------------------------------
INFO - Step 1100, rl-loss: 0.31007665395736694
INFO - Copied model parameters to target network.
INFO - Step 1103, rl-loss: 0.5912075638771057
---------------------------------------
  episode      |  198367
  reward       |  -0.1653
---------------------------------------
INFO - Step 1173, rl-loss: 0.47464087605476385
---------------------------------------
  episode      |  208703
```

```
 reward       |  -0.0803
--------------------------------------
INFO - Step 1244, rl-loss: 0.70128870010375984
--------------------------------------
 episode      |  222223
 reward       |  -0.0824
--------------------------------------
INFO - Step 1312, rl-loss: 0.49619013071060184
--------------------------------------
 episode      |  235723
 reward       |  -0.0877
--------------------------------------
INFO - Step 1372, rl-loss: 0.58226829767227176
--------------------------------------
 episode      |  4764430
 reward       |  -0.0612
--------------------------------------
--------------------------------------
 episode      |  4779824
 reward       |  -0.0584
--------------------------------------
--------------------------------------
 episode      |  4795212
 reward       |  -0.0581
--------------------------------------
--------------------------------------
 episode      |  4810230
 reward       |  -0.0574
--------------------------------------
--------------------------------------
 episode      |  4825700
```

```
 reward       |  -0.0693
--------------------------------------
--------------------------------------
 episode      |  4841193
 reward       |  -0.0599
--------------------------------------
--------------------------------------
 episode      |  4856658
 reward       |  -0.0598
--------------------------------------
--------------------------------------
 episode      |  4872287
 reward       |  -0.0614
--------------------------------------
--------------------------------------
 episode      |  4887810
 reward       |  -0.033
--------------------------------------
--------------------------------------
 episode      |  4902802
 reward       |  -0.0511
--------------------------------------
--------------------------------------
 episode      |  4918236
 reward       |  -0.0558
--------------------------------------
--------------------------------------
 episode      |  4932999
 reward       |  -0.0499
--------------------------------------
--------------------------------------
```

```
 episode      | 4947986
 reward       | -0.0396
-------------------------------------
-------------------------------------
 episode      | 4962668
 reward       | -0.0707
-------------------------------------
-------------------------------------
 episode      | 4977468
 reward       | -0.0603
-------------------------------------
-------------------------------------
 episode      | 4992969
 reward       | -0.0516
-------------------------------------
-------------------------------------
 episode      | 5008302
 reward       | -0.0762
-------------------------------------
-------------------------------------
 episode      | 5023223
 reward       | -0.0669
-------------------------------------
-------------------------------------
 episode      | 5038643
 reward       | -0.0557
-------------------------------------
-------------------------------------
 episode      | 5054143
 reward       | -0.0816
-------------------------------------
```

Aquesta gràfica mostra els valors anteriors d'una manera més visual i suposa els resultats del primer programa:

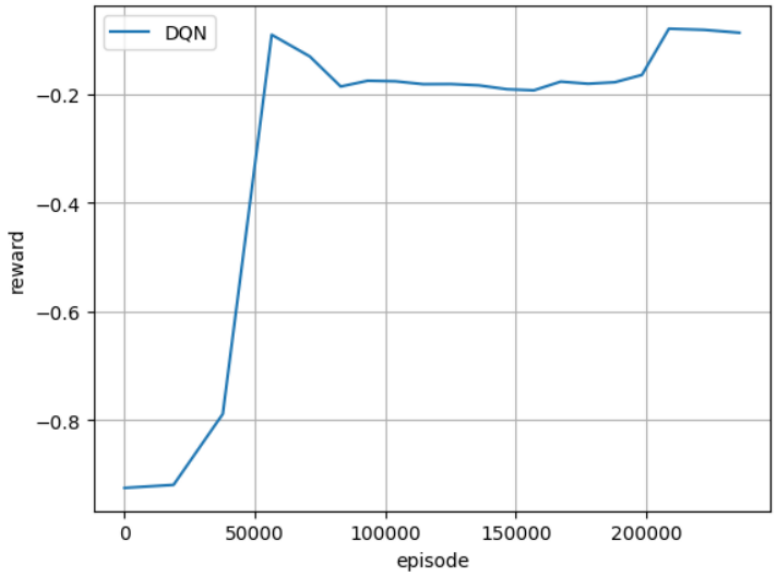


Figura 1 - Gràfic de resultats del primer programa.

# 2. MONTE CARLO - PROGRAMA DEFINITIU I COMPLET

#Codi 2 d'en Pau Amorós Faro IES Alcúdia ---- TR: Aprenetatge automàtic aplicat al Blackjack

```
import random          #Importar random, que inclou funcions per generar nombres aleatoris.
import matplotlib.pyplot as plt       #S'importa una altra llibreria, que dona la possibilitat de crear gràfics. Aquesta llibreria s'ha utilitzat al programa anterior.

DEMANAR_UNA_CARTA_MÉS = 1       #Es genera una constant amb valor 1 per a utilitzar-la més endavant.
QUEDAR_IGUAL = 0       #Es genera una constant amb valor 0 per a utilitzar-la més endavant.

class BlackJack:
  cartes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]  #Aquesta classe representa una partida de Blackjack. La classe té un únic atribut, "cartes", que és una llista que conté els valors de les cartes en el joc (és a dir, els números de l'1 al 10, així com quatre 10). Aquesta llista s'utilitzarà més tard en el codi per a simular les targetes de dibuix.

  def __init__(self, jugador_suma, crupier_visible, as_11):
    self.jugador_suma = jugador_suma          #Suma de les cartes del jugador.
    self.crupier_visible = crupier_visible          #Valor de la carta del crupier.
    self.as_11 = as_11      #Diu si es té un as que pugui contar també com a 11 o no.

  def treuCarta(self):
    return random.choice(self.cartes)    #Se simula el fet de treure una carta de la llista cartes (creada anteriorment).
```

```
  def tornCrupier(self):                #Representa el torn del crupier a una partida de blackjack.
      crupier_suma = self.crupier_visible #primer s'inicialitza la suma actual del crupier amb el valor de la carta visible per al jugador. Si aquesta carta és un as amb valor 1, es compta com 11 (com faria el crupier en una partida real de blackjack), en cas contrari, es compta com el seu valor nominal.
      if crupier_suma == 1:
          crupier_as = True
          crupier_suma = 11
      else:
          crupier_as = False

      while crupier_suma < 17:
          nova_carta = self.treuCarta()
          crupier_suma = crupier_suma + nova_carta
          if crupier_suma > 21 and crupier_as == True:
              crupier_suma = crupier_suma - 10 #La partida no ha acabat encara.
              crupier_as = False
          elif nova_carta == 1 and crupier_suma + 10 < 22:
              crupier_suma = crupier_suma + 10
              crupier_as = True          #Llavors, el crupier comença a treure cartes fins que la seva suma sigui almenys 17. Cada carta s'extreu de la baralla utilitzant el mètode "treuCarta" definit anteriorment. Si la suma del distribuïdor supera els 21, i tenen almenys un as comptat com a 11, un dels asos es canvia a un valor d'1 per evitar passar-se. Això es fa restant 10 de la suma. Si no hi ha asos comptats com a 11, el mètode simplement retorna l'estat actual del joc (incloent la suma del distribuïdor, la suma del jugador, i l'estat de l'as comptat com a 11).


      if crupier_suma > 21:
```

```
            return [self.jugador_suma, self.crupier_visible, self.as_11, 1, True]
        if crupier_suma > self.jugador_suma:
            return [self.jugador_suma, self.crupier_visible, self.as_11, -1, True]
```

*#Es verifica si la suma de les cartes del crupier és major que la suma de les cartes del jugador, però el crupier no s'ha passat de 21. Si aquesta condició és veritable, significa que el crupier té una mà millor que el jugador sense passar-se de 21, per la qual cosa el jugador perd. La funció retorna una llista similar a l'anterior, però amb un valor de -1 per a indicar que el jugador va perdre.*

```
        if self.jugador_suma > crupier_suma:
            return [self.jugador_suma, self.crupier_visible, self.as_11, 1, True]
```

*#Aquesta condició verifica si la suma de les cartes del jugador és major que la suma de les cartes del crupier, i el crupier no s'ha passat de 21. Si aquesta condició és veritable, significa que el jugador té una mà millor que el crupier sense passar-se de 21, per la qual cosa el jugador guanya. La funció retorna una llista similar a la primera, amb un valor d'1 per a indicar que el jugador ha guanyat.*

```
        if self.jugador_suma == crupier_suma:
            return [self.jugador_suma, self.crupier_visible, self.as_11, 0, True]
```

*#Aquesta condició verifica si la suma de les cartes del jugador és igual a la suma de les cartes del crupier, la qual cosa resulta en un empat. La funció retorna una llista similar a la primera, però amb un valor de 0 per a indicar que la partida ha acabat en empat.*

```
    def torn(self, acció):
        if acció == QUEDAR_IGUAL:
            return self.tornCrupier()
        else:
            nova_carta = self.treuCarta()
```

```
        self.jugador_suma = self.jugador_suma + nova_carta

        if self.jugador_suma > 21 and self.as_11 == True:
            self.jugador_suma = self.jugador_suma - 10
            self.as_11 = False

        elif nova_carta == 1 and self.jugador_suma + 10 < 22:
            self.jugador_suma = self.jugador_suma + 10
            self.as_11 = True

        if self.jugador_suma > 21:
            return [self.jugador_suma, self.crupier_visible, self.as_11, -1, True]
        else:
            return [self.jugador_suma, self.crupier_visible, self.as_11, 0, False]
```
*#Finalment, el mètode retorna el resultat del torn del distribuïdor en funció de la suma final. Si la suma del repartidor supera els 21, el jugador guanya. Si la suma del distribuïdor és més gran que la del jugador, el jugador perd. Si la suma del jugador és més gran que la del repartidor, el jugador guanya. Si el jugador i el distribuïdor tenen la mateixa suma, és taules. El mètode retorna una llista de valors, incloent-hi l'estat actual del joc i una bandera que indica si el joc s'ha acabat o no.*

```
class Estat:    #Classe explicada amb profunditat al treball.

   def __init__(self, jugador_suma, crupier_visible, as_11):     #Tres atributs explicats anteriorment. El mètode "init" s'encarrega d'inicialitzar totes les variables, assignant als comptadors una freqüència inicial d'1 i a les recompenses totals el valor de la recompensa passada com a paràmetre.

      self.jugador_suma = jugador_suma
      self.crupier_visible = crupier_visible
      self.as_11 = as_11
```

```
        self.n_cartamés = 1
        self.n_quedarseigual = 1
        self.Q_cartamés_total = 0
        self.Q_quedarseigual_total = 0
        self.política = QUEDAR_IGUAL

    def actualitza(self, recompensa, acció):          #La funció "actualitza" pren dos
```
*arguments: «recompensa» i «acció». «Recompensa» és un número que representa la recompensa rebuda després de realitzar una acció en un estat. Daltra banda, «acció» és una variable que indica l'acció que es va prendre en aquest estat («DEMANAR_UNA_CARTA_MÉS» o «QUEDAR_IGUAL»).*

```
        if acció == QUEDAR_IGUAL:
            self.n_quedarseigual = self.n_quedarseigual + 1
            self.Q_quedarseigual_total = self.Q_quedarseigual_total + recompensa
        else:
            self.n_cartamés = self.n_cartamés + 1
            self.Q_cartamés_total = self.Q_cartamés_total + recompensa
```

*#S'actualitzen valors amb els nous valors de la recompensa i el nombre de vegades que s'ha pres una acció determinada.*

```
        if self.Q_cartamés_total / float(self.n_cartamés) > self.Q_quedarseigual_total /
float(self.n_quedarseigual):
            self.política = DEMANAR_UNA_CARTA_MÉS
        else:
            self.política = QUEDAR_IGUAL
```
*#Finalment, s'actualitza el valor de «política» segons la regla "epsilon-greedy". Es compara la recompensa mitjana de prendre una carta addicional amb la recompensa mitjana de quedar-se igual. Si la recompensa mitjana de prendre una carta addicional*

*és major, llavors la política s'actualitza a «DEMANAR_UNA_CARTA_MÉS», en cas contrari, la política s'actualitza a «QUEDAR_IGUAL».*

```
def determinaEstat(jugador_suma, crupier_visible, as_11):
  return ((jugador_suma - 11) * 10 + crupier_visible) * 2 - as_11 - 1
```

*#Aquesta funció llavors calcula un índex que representa l'estat del joc utilitzant la fórmula: ((jugador_suma - 11) * 10 + crupier_visible) * 2 - as-11- 1*
*Això significa el següent:*

- *(jugador_suma - 11): Aquesta part resta 11 de la suma del jugador per a fer l'interval de valors de 0 a 9 en lloc d'11 a 20.*
- ** 10: Multiplica el resultat anterior per 10 per a obtenir valors de 0 a 90 en lloc de 0 a 9.*
- *+ crupier_visible: afegeix el valor de la targeta visible del distribuïdor al resultat anterior.*
- ** 2: Multiplica el resultat anterior per 2 per a aconseguir valors de 0 a 180 en lloc de 0 a 90.*
- *- as_11 - 1: resta el valor d' as_11 i 1 del resultat anterior. Aquesta part de la fórmula s'utilitza per diferenciar entre els dos valors possibles d'as (011 (0 o 1).*

*El resultat final de la fórmula és l'índex que representa l'estat actual del joc. Aquest índex s'emprarà per accedir a l'element corresponent en una matriu que conté informació sobre l'estat.*

```
def monteCarloES(num_episodis=5000000):
```
*#Explicat amb detall al treball. Aquesta funció implementa l'algoritme Monte Carlo Exploring Starts per aprendre la política òptima per jugar al blackjack. La funció pren també un argument nombre_episodis, que especifica el nombre d'episodis (jocs) a simular. El valor per defecte és 7,500,000.*

```
    estats = [Estat(i, j, l) for i in range(11, 22) for j in range(1, 11) for l in
reversed(range(2))]
```

*#Primer, la funció crea una llista d'estats que contenen tots els estats possibles del joc. Cada estat està representat per un objecte Estat, que emmagatzema la suma del jugador, la carta visible del crupier i si el jugador té un as que es pot comptar com 11.*

```
    for i in range(0, num_episodis):
```
*#A continuació, la funció entra en bucle la quantitat de vegades determinada per "nombre_episodis". Per a cada episodi, se selecciona un estat inicial aleatori "s" de la llista d'estats. Una instància de la classe Blackjack es crea amb l'estat inicial, i la primera acció d'"Acció" s'escull aleatòriament (0 o 1. Determinats anteriorment).*

```
        s = random.choice(estats)
        episodi = []
        bj = BlackJack(s.jugador_suma, s.crupier_visible, s.as_11)
        acció = random.randint(0, 1)
        episodi.append([s, acció])
```
*#La funció llavors simula el joc emprant repetidament el mètode "torn()" de la instància BlackJack fins que el joc finalitza. Cada vegada, l'estat actual "s" s'actualitza en funció de la suma del jugador, la cara visible del crupier, i si el jugador té un as que es pot comptar com 11. L'acció òptima per al nou estat es tria basant-se en l'atribut de política de l'objecte Estat. L'estat actual i l'acció escollida s'afegeixen a la llista d'episodis.*

```
        while True:
            jugador_suma, crupier_visible, as_11, recompensa, joc_finalitzat = bj.torn(acció)
            if joc_finalitzat == False:
                s = estats[determinaEstat(jugador_suma, crupier_visible, as_11)]
                acció = s.política
                episodi.append([s, acció])
            else:
                for e in episodi:
```

```
                e[0].actualitza(recompensa, e[1])
            break
    return estats
```

*#Una vegada que el joc acaba, la funció es repeteix sobre la llista d'episodis i actualitza els valors Q de cada parell d'acció i estat utilitzant el mètode "actualitza()" de l'objecte "Estat". Finalment, després que tots els episodis s'hagin simulat, la funció retorna la llista d'estats, que ara conté la política òptima apresa per jugar a blackjack.*

```
def determinaPolítica(estats):        #Agafa la llista d'estats com a "input" i crea dues
    cartamés_no_as = dict()           #1r diccionari.
    cartamés_as = dict()        #2n diccionari.
    for s in estats:
        if s.política == DEMANAR_UNA_CARTA_MÉS and s.as_11 == False: #El jugador no demana una carta més.

            if s.crupier_visible in cartamés_no_as:
                cartamés_no_as[s.crupier_visible] = max(cartamés_no_as[s.crupier_visible], s.jugador_suma)
            else:
                cartamés_no_as[s.crupier_visible] = s.jugador_suma
        elif s.política == DEMANAR_UNA_CARTA_MÉS\
                and s.as_11 == True:
            if s.crupier_visible in cartamés_as:
                cartamés_as[s.crupier_visible] = max(cartamés_as[s.crupier_visible], s.jugador_suma)
            else:
                cartamés_as[s.crupier_visible] = s.jugador_suma
```

*#Els diccionaris s'utilitzen per determinar l'estratègia a seguir en el joc. Si la suma actual del jugador està per sota del valor corresponent en el diccionari per a la carta visible actual, el jugador hauria de demanar una altra carta. En cas contrari, el jugador s'hauria de quedar.*

```
xrange = range     #Necessari en la versió de Python que s'ha emprat.

lists = sorted(cartamés_no_as.items())
x, y = zip(*lists)     #Defineix les llistes de variables com una llista ordenada
```

*d'elements del diccionari nodemanar_amb_as. Cada element de la llista és una tupla que conté un parell clau-valor del diccionari. La funció zip s'utilitza per a separar les claus i els valors de cada tupla a la llista de llistes i assignar-los a variables separades x i y. Això crea dues llistes, x i y, on x conté les claus del diccionari nodemanar.amb.as i y conté els valors corresponents. En resum, aquest bloc de codi està preparant les dades del diccionari nodemanar.amb.as per a la visualització o anàlisi posterior. Ordena els elements del diccionari per clau i crea dues llistes, x i y, per mantenir les claus i valors ordenats, respectivament.*

```
plt.figure(figsize=(12, 6)) #Mesura del gràfic
plt.subplot(1, 2, 1)          #Aquest codi utilitza la biblioteca Matplotlib per a crear una
```

*visualització. Crea un subplot amb 1 fila, 2 columnes, i selecciona la primera columna per a dibuixar les dades.*

```
plt.step(x, y, where='mid')                #Les dades es proporcionen com dues llistes
```

*de valors x i y, que es comprimeixen junts utilitzant la funció zip. Aquestes llistes representen els punts que es dibuixaran al gràfic. La funció plt.plot() es crida per a dibuixar les dades com una funció de pas, on el paràmetre WHERE s'estableix a «mitjà» per a dibuixar els passos al punt mig entre cada valor x.*

```
  eixos = plt.gca()
  eixos.set_ylim([10, 22])
  eixos.yaxis.set_ticks(xrange(10, 23, 1))
  eixos.xaxis.set_ticks(xrange(1, 10, 1))
  plt.xlabel("Visible crupier")
  plt.ylabel("Total jugador")
  plt.title("L'AS TÉ VALOR 11", fontsize=14)
  plt.text(0.5, 0.8, "QUEDAR-SE IGUAL", fontsize=12, horizontalalignment="center",
transform=eixos.transAxes)
  plt.text(0.8, 0.2, "DEMANAR UNA CARTA", fontsize=12, horizontalalignment="center",
transform=eixos.transAxes)          #En conclusió, aquest codi està traçant un gràfic que
mostra l'estratègia òptima per a un jugador en un joc de blackjack, on el jugador pot
optar per quedar-se o colpejar en funció de la seva mà actual i la carta visible del
distribuïdor. El gràfic mostra el valor de la mà del jugador en l'eix Y, la targeta visible del
distribuïdor en l'eix X, i l'acció òptima a prendre (posar-se o prémer) s'indica per la
funció de pas.


  lists = sorted(cartamés_as.items())      #La funció "sorted" s'utilitza per a ordenar el
diccionari de cartamés_as, que conté el nombre de la targeta visible del distribuïdor i la
suma màxima que el jugador hauria d'obtenir si decideix demanar una altra carta en lloc
de quedar-se igual.


  x, y = zip(*lists)     #Aquest codi crea un gràfic utilitzant Matplotlib per mostrar
l'estratègia recomanada per a un jugador de Blackjack quan té un as a la mà amb un
valor d'11.


  plt.subplot(1, 2, 2)
  plt.step(x, y, where='mid')
  eixos = plt.gca()
  eixos.set_ylim([10, 22])
```

```
    eixos.yaxis.set_ticks(xrange(10, 23, 1))
    eixos.xaxis.set_ticks(xrange(1, 10, 1))
    plt.xlabel("Visible crupier")        #Títol eix x.

    plt.ylabel("Total jugador")          #Títol eix y.

    plt.title("L'AS TÉ VALOR 1", fontsize=14)       #Títol general i tamany que tindrà.
    plt.text(0.5, 0.8, "QUEDAR-SE IGUAL", fontsize=12, horizontalalignment="center",
transform=eixos.transAxes)          #Configuració de l'aspecte.
    plt.text(0.5, 0.2, "DEMANAR UNA CARTA", fontsize=12, horizontalalignment="center",
transform=eixos.transAxes)
    plt.show()   #Es representa el gràfic.




if __name__ == "__main__":
        estats = monteCarloES()
        determinaPolítica(estats)
```

*#Aquesta part del codi s'utilitza per executar l'algorisme de Monte Carlo per simular el joc de Blackjack i determinar l'estratègia òptima per al jugador. La funció monteCarloES() executa l'algorisme de Monte Carlo per generar un conjunt d'estats del joc i els seus corresponents retorns esperats. La funció "determinaPolítica(estats)" pren aquest conjunt d'estats de joc i els seus retorns esperats com a entrada i determina l'estratègia òptima per al jugador en cada estat de joc possible. El bloc if .name = == "_main_": és un llenguatge Python comú que permet que el codi que conté només s'executi quan el fitxer s'executa com un programa independent, en lloc de quan s'importa com un mòdul en un altre programa.*

# 2.1 RESULTATS

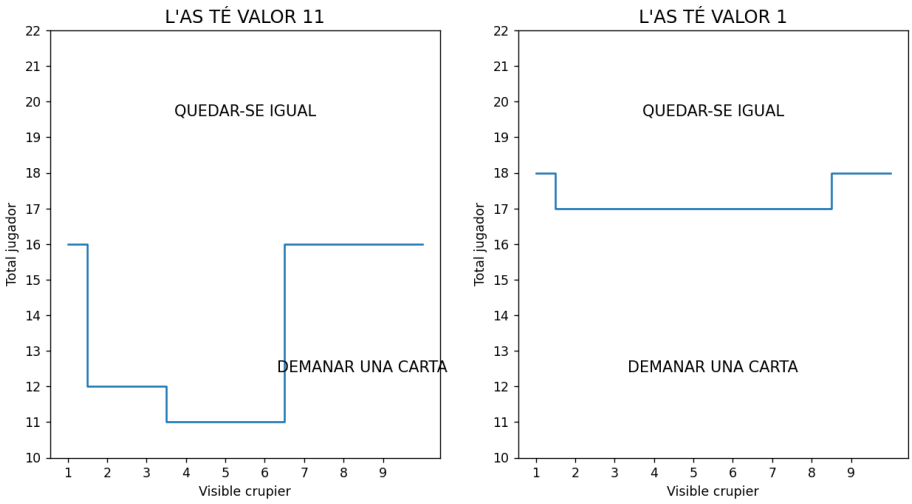


Figura 14 - Gràfic de resultats del primer programa.

Aquesta és l'aproximació del segon programa a què seria l'estratègia més bona. Així doncs, es pot realitzar una comparació amb l'estratègia de joc estandarditzada. Arran d'aquesta comparació, es pot determinar que el codi està ben redactat, car l'estructura de l'estratègia és menys o menys igual. Això no obstant, hi ha lleugeres discrepàncies. Per llegir els gràfics es faria el següent. En primer lloc, es miraria el total del jugador. A continuació la carta visible del crupier. Si es dona el cas que la intersecció d'aquests dos valors està damunt la línia blava, llavors el jugador ha de quedar-se igual. Si passa el contrari, demanar una carta més.

# 3. GLOSSARI DELS MÈTODES UTILITZATS

L’adequat funcionament dels dos programes desenvolupats en aquest annex no seria possible sense l’ús de mètodes de diferents llibreries. Els més freqüents són:

make(): Aquesta funció és generalment utilitzada per crear un entorn de simulació. Pot ser usada en problemes d'intel·ligència artificial per crear una situació de prova o entrenament.

num_actions(): Aquest mètode és fet servir per obtenir el nombre d'accions que es poden dur a terme en un determinat entorn o situació. Això pot ser útil en la creació d'algoritmes de presa de decisions.

state_shape[]: Aquesta variable conté la forma o dimensions de l'estat en què es troba un agent en un determinat moment. És una informació important per a la presa de decisions per part de l'agent.

set_agents(): Aquesta funció és utilitzada per establir el nombre d'agents que interactuen en un entorn determinat. Això pot ser útil per a la creació de simulacions d'interacció entre múltiples agents.

run(): Aquest mètode és emprat per executar un bucle de simulació en un entorn determinat. Això pot ser útil per a la creació d'algorismes d'aprenentatge reforçat.feed(): Aquest mètode és usat per alimentar l'entorn amb les accions realitzades per l'agent. Això permet actualitzar l'estat actual de l'entorn i continuar la simulació.

log_performance(): Aquest mètode és fet servir per registrar les dades de rendiment d'un agent durant la simulació. Això pot ser útil per avaluar la seva eficàcia i millorar la seva capacitat de presa de decisions.

csv_path(): Aquesta variable conté la ruta d'accés a un arxiu CSV, que és un format de fitxer comúment utilitzat per emmagatzemar dades tabulars.

fig_path(): Aquesta variable conté la ruta d'accés a un arxiu d'imatge, que pot ser útil per emmagatzemar gràfics o visualitzacions generades durant la simulació.

self(): Aquesta paraula clau es refereix a l'objecte actual que estem manipulant en un programa orientat a objectes.

policy(): Aquesta variable conté la política de decisió que està utilitzant un agent per prendre decisions en un determinat moment. Això pot ser útil per avaluar el seu rendiment i millorar la seva capacitat de presa de decisions.

random.choice(): Aquesta funció és feta servir per seleccionar un element aleatori d'una llista o conjunt de dades.

random.randint(): Aquesta funció és usada per generar un nombre enter aleatori dins d'un rang donat.

append(): Aquest mètode és utilitzat per afegir un element a una llista o a una altra estructura de dades.

plt.figure(): Aquesta funció és emprada per crear una figura en blanc per a la generació de gràfics.
plt.show(): Aquesta funció és usada per mostrar un gràfic generat amb la biblioteca Matplotlib. Sense aquesta funció, el gràfic no es mostraria en la pantalla i l'usuari no podria veure'n els resultats.