كلية الهندسة المعلوماتية

السنة الرابعة – ذكاء صنعي

مشروع عملي خوارزميات البحث الذكية

لعبة اللودو

إعداد الطلاب:

آيات أحمد الحلواني

إيمان محمد جمال بازر

غنى عامر الحرش

غنى زاهر الرفاعي

عمر لؤي الميداني

علي محمد زيتون

بإشراف: د. عمار النحاس

23/1/2025

الفهرس

المقدمة

اللعبة:

اللّودو هي واحدة من أكثر الألعاب اللوحية شهرة وانتشارًا في جميع أنحاء العالم. تعود جذور هذه اللعبة إلى الهند، حيث تطورت من لعبة قديمة تُعرف باسم "باچايسي". تتمحور فكرة اللعبة حول الحركة الاستراتيجية، حيث يتنافس اللاعبون على إدخال أربعة قطع إلى "المنزل" الخاصة بهم، عن طريق التقدم في مسار دائري يشتمل على 52 مربعًا.

الهدف من اللعبة:

الهدف الرئيسي في لعبة اللودو هو أن يكون اللاعب أو الفريق أول من يُدخل جميع القطع إلى "المنزل" الخاصة به، متغلبًا بذلك على باقي اللاعبين. تعتمد اللعبة على الحظ في رمي النرد، بالإضافة إلى الاستراتيجية في تحريك القطع والتفاعل مع اللاعبين الآخرين.

عدد اللاعبين:

تُعتبر اللودو لعبة جماعية، حيث يمكن أن تُلعب بواسطة 2 إلى 4 لاعبين، وهناك نسخ مختلفة من اللعبة يمكن أن تستوعب عددًا أكبر من اللاعبين.

طريقة اللعب:

تبدأ اللعبة بوضع جميع القطع في مناطق البداية الخاصة بكل لاعب. يتم تحديد اللاعب الذي سيبدأ أولاً وفق نظام معين، يتمثل عادة في رمي النرد. يتحرك اللاعب وفقًا للعدد الذي يظهره النرد، ويمكنه استخدام هذه الحركة لتحريك القطع في الميدان أو لإخراج قطع خصومه من اللعبة. يعتبر توجيه الاستراتيجيات المميزة والتخطيط لتحركات الخصوم جزءًا مهمًا من نجاح اللاعب في الفوز.

الأبعاد الاجتماعية والثقافية:

تُعتبر لعبة اللودو وسيلة ترفيه عائلية ممتعة، حيث تساهم في تعزيز الروابط الاجتماعية بين الأصدقاء والعائلة. تمتاز بجو من المرح والمنافسة الودية، مما يجعلها خيارًا شائعًا في المناسبات الاجتماعية والاحتفالات.

بنية اللعبة

بدايةً اللعبة يمكن تمثيلها بمصفوفة أحادية لأنها ذات مسار واحد.

قمنا بمراعاة أماكن بدأ كل لاعب وأماكن دخوله لمسار فوزه الخاص وذلك يكون حسب رقم اللاعب حيث أن كل لاعب يكون مكان بدئه الخاص بعد مكان بدأ اللاعب الذي يسبقه.

أيضاً قمنا بمراعاة انتقال كل لاعب مكان بدأه ليس في الموضع 0 حيث أنه عندما يصل إلى الخانة 51 ينتقل مباشرة بعدها إلى الخانة صفر (أي أن المصفوفة أصبحت دائرية).

وعند تفكيرنا بالانتقال وقتل اللاعبين لأحجار بعضهم وجدنا أنه يضعف أداء البنية أن يمر على أحجار جميع اللاعبين ومعرفة أماكنهم ومن ثم معالجة القتل. فقمنا بتخزين string في المصفوفة الأحادية تحتوي على ثنائيات العنصر الأول من الثنائي يدل على اللاعب والعنصر الثاني يدل على رقم الحجر الخاص به فأصبح التأكد من حالة القتل أسرع.

بالإضافة إلى ذلك وجدنا أنه عند رمي النرد واختيار اللاعب لحجره من الصعب التعامل فقط مع المصفوفة الأحادية حيث

أنه سيصبح تعقيد التأكد من إمكانية تحريك كل حجر له كبيرة جداً لأننا سوف نقوم بالمشي على المصفوفة الأحادية كاملة وعلى كل حجر في كل موضع لذلك قررنا إضافة مصفوفة مساعدة تدل على مكان كل حجر لكل لاعب في الرقعة (pieces) يوضع فيها موضعه في الرقعة إذا وجد و -1 في حال كان لم يدخل الرقعة بعد و-2 في حال كان قد دخل مسار فوزه.

ولمعالجة مسارات الفوز وعمقها قمنا بإضافة مصفوفة أحادية تدل أن لكل لاعب العمق التالي لمسار فوزه هو maximum_win_depth[i].

كل من البنى السابقة تم وضعها في كلاس state واستخدمنا object منه في كلاس الnode وإضافة إلى ذلك تضمن كلاس النود عداد لعمليات القتل وعداد لعدد نتائج النرد المساوي للقيمة ستة بشكل متتالي وذلك لتفيدنا في تحديد اللاعب القادم من خلال تابع WhoIsNextPlayer.

ومصفوفة chance لحساب كل رقعة ما قيمة فائدتها لكل لاعب سنقوم بشرحها لاحقا في خوارزمية expectminmax وأخيراً مصفوفة الحركات القادمة nextnodes.

معرفة الحركات الممكنة

لتوليد العقد القادمة نحتاج معرفة الحركات الممكنة القادمة وهنا بدأنا النقاش بكيفية توليدهم ثم قررنا تقسيم الحالات التي يمكن الحركة فيها إلى:

1- أي قيمة نرد عدا الستة للأحجار المركبة مرفوضة
2- في حال كان الحجر قد دخل مسار الفوز فجميع حركاته مرفوضة
3- بعد أن تأكدنا أن الحجر داخل الرقعة الآن نتأكد من كل قيمة نرد على حدى على أساس الحالات الآتية:
   a. في حال كان الموقع الجديد هو عبارة عن حائط فهذه أقصى قيمة يمكن للاعب أن يتحرك بعدها
   b. التحقق من أن الموقع الجديد لا يدخل اللاعب في مسار الفوز فهي حركة ممكنة
   c. في حال كان الموقع الجديد هو داخل مسار الفوز فنتأكد إن كان يحقق العمق المناسب: فإذا كان كذلك فهذه حركة ممكنة.

وعدا جميع الحالات السابقة فهي حركات غير ممكنة.

```python
def Is_Wall(self, Pos , player):
    if len(self.state.board[int(Pos)]) < 4:
        return False
    cnt = np.full(4 , 0)
    wall = False
    for i,j in enumerate(self.state.board[int(Pos)]):
            if i%2 == 0 and j != 's':
                cnt[ord(j) - ord('0')] += 1
                if cnt[ord(j) - ord('0')] >= 2 and ord(j) - ord('0') != player :
                    wall = True
    return wall
```

```python
def Is_Win_Path(self, player, Pos , dice):
    end_pos = 0
    if(player == 0):
        end_pos = 50
    else:
        end_pos = (player) * 13 - 2
    if end_pos >= Pos and dice - (end_pos - Pos) > 0:
        return dice - (end_pos - Pos)
    return -1
```

```python
def Get_Possible_Moves(self , nextPlayer):
    possible_moves = [] # عم تخزن بيرات كل بير بيعبر عن رقم القطعة وعدد الخطوات يلي عم تقدر تتحركهم
    for i in range(4): # عم نمرق على القطع
        if self.state.piece[nextPlayer][i] == -1: # اذا القطعة لسا ما تركبت
            possible_moves.append((i , 6))
        elif self.state.piece[nextPlayer][i] == -2: # اذا فازت القطعة ف مافي انها ايا حركة
            continue
        else:
            for j in range(1 , 7): # عم نمرق عل الحركات
                pos = self.state.get_pos(self.state.piece[nextPlayer][i] + j) # لنجيب البوزيشن الجديد بعد ما طبقنا الخطوات
                # عم نتاكد اذا القطعة بتوصل لمسار فوز اما لا و التابع بيرجعلي كم خطوة مشيت بمسار الفوز اذا فتت و -1 اذا ما فتت
                win_pos = self.Is_Win_Path(nextPlayer, self.state.piece[nextPlayer][i], j)
                if self.Is_Wall(pos, nextPlayer): # عم نتاكد اذا في جدار بعد ما طبق الخطوات
                        if self.state.board[pos][-1] != 's':# اذا في جدار ف بدي اتاكد اذا هو بمنطقة امنة اما لا
                            possible_moves.append((i , j))# اذا هو بمنطقة امنة ف بقدر وقف عل الجدار
                        break
                elif win_pos==-1:# اذا مافي شي بالمكان الجديد يلي رايح عليه ف خلص الحركة ممكنة
                    possible_moves.append((i , j))
                elif self.state.maximum_win_depth[nextPlayer] == win_pos: # اذا وصل لاعمق نقطة بمسار الفوز
                    possible_moves.append((i , j))

    return possible_moves

```

توليد العقد القادمة

الآن ننتقل لتوليد العقد القادمة وذلك بناءً على:

- من هو اللاعب القادم
- وعلى حركاته الممكنة.

نعالج جميع حالات الإزالة من المكان القديم والإضافة للمكان الجديد، وفي حال كان داخل الرقعة نقوم بالتأكد من حالة قتل الأحجار و تقليل عمق مسار الفوز للاعب في حال دخل لمسار الفوز.

إلى هنا كنا قد انتهينا من مناقشة بنية اللعبة وقرنا البدء بتكويدها والتاكد من ان جميع الحالات محققة بشكل جيد.

```
def Generate_Next_States(self):

    new_player=self.WhoIsNextPlayer()
    P_actions=self.Get_Possible_Moves(new_player)

    for i in P_actions:
        new_node=Node(self.playersNum)
        new_node.deepCopy(self, new_player)

        if i[1] == 6:
            if new_node.six_hit_counter<3:
                new_node.six_hit_counter=new_node.six_hit_counter+1
            else:
                new_node.six_hit_counter=0
        else:
            new_node.six_hit_counter=0

        curr_substr = str(new_player) + str(i[0])

        if new_node.state.piece[new_player][i[0]]==-1 and i[1] == 6:    #تركيب القطعة
            new_node.state.piece[new_player][i[0]]=new_player*13
            new_pos=new_player*13
            new_node.state.board[new_pos]=curr_substr+new_node.state.board[new_pos]   #add the piece to the new position string
            new_node.chance[new_player]+=7

        else:
            old_pos=new_node.state.piece[new_player][i[0]]
            curr_ind = new_node.state.board[old_pos].find(curr_substr)
            new_node.state.board[old_pos] = new_node.state.board[old_pos][:curr_ind] + new_node.state.board[old_pos][curr_ind+2:]  # remove the peice from the string of the old position

            win_pos = self.Is_Win_Path(new_player, old_pos, i[1])
            if win_pos != -1:
                new_node.state.piece[new_player][i[0]] = -2
                new_node.state.maximum_win_depth[new_player]=new_node.state.maximum_win_depth[new_player]-1 #نقصت العمق تبع مسار الفوز
                new_node.chance[new_player]+=25


            else:
                new_pos=new_node.state.get_pos( old_pos+i[1])
                new_node.state.piece[new_player][i[0]] = new_pos
                new_node.chance[new_player] += i[1]
                for j in range(len(new_node.state.board[new_pos])):
                    if j%2==1:
                        continue
                    if new_node.state.board[new_pos][j]!=str(new_player) and self.state.board[new_pos][-1] != 's': #kill the Opponent
                        add =0
                        died = ord(new_node.state.board[new_pos][j])-ord('0')
                        if (died * 13 <= new_pos):
                            add = new_pos - died*13 + 1
                        else:
                            add = 52 - died*13 + new_pos
                        new_node.chance[new_player]+=add/2
                        new_node.chance[died]-=(add + 9)
                        new_node.state.piece[ord(new_node.state.board[new_pos][j])-ord('0')][ord(new_node.state.board[new_pos][j+1])-ord('0')]=-1 #change its place in the piece array
                        new_node.state.board[new_pos]=new_node.state.board[new_pos][:j]+new_node.state.board[new_pos][j+2:] #delete it from the string in bpard array
                        new_node.killCnt=new_node.killCnt+1
                        j=j-1


                new_node.state.board[new_pos]=curr_substr+new_node.state.board[new_pos]   #add the piece to the new position string
                if new_node.killCnt==self.killCnt: new_node.killCnt=0
                else:  new_node.killCnt=1

        self.nextnodes[i[0]][i[1]]=new_node

        if new_player!=self.player:
            for i in range(0,4):
                for j in range(1,7):
                    if self.nextnodes[i][j]==None:
                        new_node=Node(self.playersNum)
                        new_node.deepCopy(self, new_player)
                        if j==6: new_node.six_hit_counter=new_node.six_hit_counter+1
                        self.nextnodes[i][j]=new_node
    return P_actions
```

خوارزمية Expectminmax:

بعد تأكدنا من بنية اللعبة بدأنا بالتفكير باستراتيجية منطقية ليتم اتخاذ قرار التحريك للحاسب من خلالها ووضعنا النقاط المبدئة الحالية:

1- يهمنا أن يكون إضافة حجر جديد للعبة أكثر أهمية من التحرك العادي لأي حجر في الرقعة لذلك أعطيناه قيمة أكبر من أي تحرك وهي 7.

2- كل تحرك عادي يُعطي نقاط بعدد الحركة لأن ذلك يُقربنا من الهدف.

3- عند قتل حجر خصم يأخذ اللاعب نقاط بعدد حركته تقسيم اثنان لأن ذلك سيفيدنا بإبعاد الخصم عن الفوز لكنه لن يفيدنا بمقدار حركتنا لذلك قسمناه على اثنان.

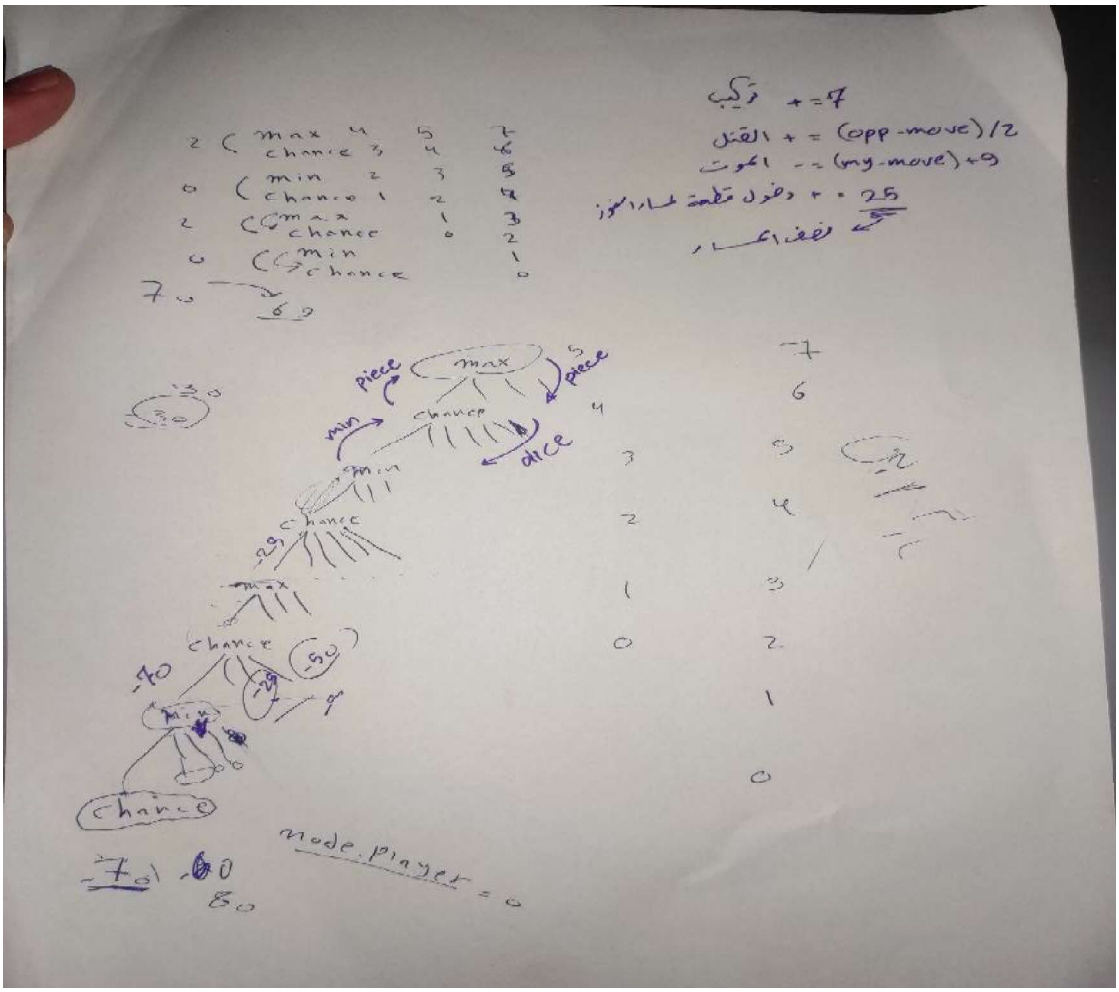
4- عند خسارة حجر سيخسر اللاعب جميع الحركات التي تحركها وقمنا بإضافة خسارة نقطتان لضياع الوقت.

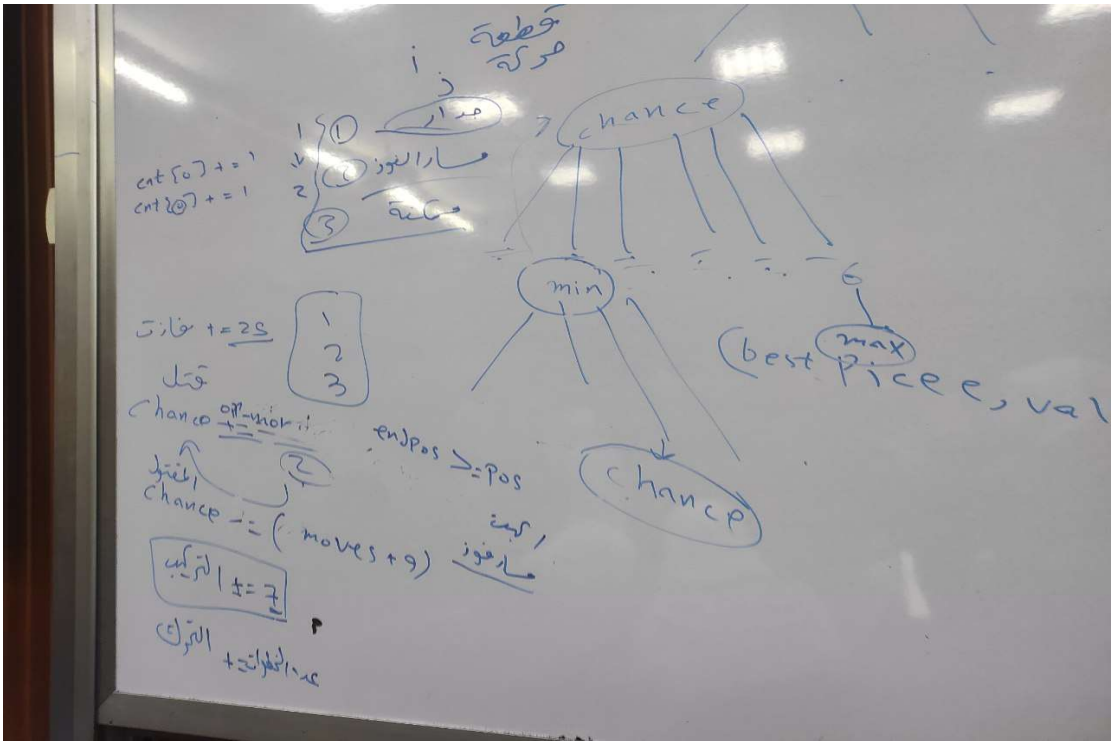
وبعد وضع توزيع النقاط اتجه التفكير إلى كيفية حساب هذه النقاط لكل رقعة وهنا ظهرت فكرة مصفوفة ال chances حيث أننا بعد دراستنا للخوارزمية وجدنا أنه في عملية ال evaluate عند الأوراق، الشجرة سيكون عندها نقطة اتخاذ القرار الأولية والعمليات التي ستجري عليها مكلفة جدا لكنها لحد ما تشبه العمليات المطبقة في Generate_Next_States.

فعدنا لفكرة dynamic programming التي أخذناها سابقاً في مقرر مادة الخوارزميات حيث أنها ستقلل من التعقيد بشكل كبير لأن فائدة العقد سيكون قد تم حسابها بشكل تراكمي وسابق لوصولنا للعقدة حتى فأصبحنا عند توليد العقدة نحسب تكلفة وفائدة الحركة المنفذة حالياً ونضيفها إلى التكاليف السابقة لهذا اللاعب.

وهكذا أصبحت لدينا الفوائد للعقد وبقي بناء الخوارزمية.

انطلقنا في بناء الخوارزمية من فكرة الحظ والقرار حيث أن تحريك الحجر بحد ذاته هو قرار وقيمة النرد هو الحظ ولكن القرار يجب أن يبنى على الحظ لذلك قمنا بعملية تناوب بين عقد الحظ وعقد القرار (min, max) واعتبرنا أن الحاسب هو ال max وعند الوصول إلى الاوراق سنقوم بحساب المتوسط الحسابي للفوائد لكل العقد القادمة وجعلها قيمة سالبة إذا كانت الورقة تابعة للاعب البشري، وتركها قيم موجبة في حال كان اللاعب هو الحاسوب.






```python
def expectminmax(self, turn, node : Node , chance , depth , piece , dice):
        P_Actions = node.Generate_Next_States()
        if depth == 0 or node.Is_Win == True:
            ev = 0
            cnt = 0
            for i in range(1 , 7):
                if node.nextnodes[piece][i]!= None:
                    cnt += 1
                    ev += node.nextnodes[piece][i].chance[node.player] # think again
            temp = -100000
            if cnt > 0: temp = ev / cnt
            if node.player == 0: temp *= -1
            return (piece ,  temp)
        expected = 0
        value = -1000000000
        BestPiece = -1
        if turn == 0: value *= -1
        if chance == False:
            for i in range(4):
                can = False
                for j in P_Actions:
                    if j[1] == dice and j[0] == i:
                        can = True
                if can == True:
                    temp = self.expectminmax(turn , node , True , depth-1 , i , dice)
                    if (turn == 2 and temp[1] > value) or (turn == 0 and temp[1] < value):
                        value = temp[1]
                        BestPiece = i
            return (BestPiece , value)
        cnt = 0
        for i in range(1,7):
            if node.nextnodes[piece][i]!= None:
                temp = self.expectminmax(node.WhoIsNextPlayer() , node.nextnodes[piece][i] , False , depth-1 , piece , i)
                expected += temp[1]
                cnt += 1
        temp = -100000
        if cnt > 0: temp = expected / cnt
        if node.player == 0: temp *= -1
        return (piece , temp)
```

مخطط الصفوف


Main
Logic
-visited
+Pieces_To_Move()
+Play()
+PlayHuman()
+PlayComputer()
+expectminmax()
Node
-state
-nextnodes
-chance
-killCnt
six_hit_counter
-playersNum
player
+Generate_Next_States()
+Is_Wall()
+Is_Win_Path()
+Get_Possible_Moves()
+deepCopy()
+Apply_Move()
+Is_Win()
+WhoIsNextPlayer()
+Print_State()
State
maximum_win_depth
board
+get_pos()