به نام خدا

درس هوش مصنوعی و سیستم های خبره

تمرین ششم

مدرس : دکتر محمدی

دانشجو : سارا سادات یونسی / ۹۸۵۳۳۰۵۳

بخش تئوری

۱.دو روش برای استنتاج منطق(Inference Logical)r نام ببرید. هر روش را معرفی کرده و مثالی بزنید.

جواب الف)

ما دو روش داریم که عبارتند از : ۱.model-checking   ۲.Theorem-proving

روش اول :

- For every possible world, if α is true make sure that is β true too
- OK for propositional logic (finitely many worlds)

برای این حالت همه ی حالات موجود را چک می کنیم که اگر آلفا درست باشد آنگاه می توان گفت بتا درست است یا خیر یا به عبارت دیگر enties می کند یا خیر.

این روش برای محیط ها و مدل های محدود مناسب است زیرا در فضای بزرگتر چک کردن حالات با تعداد بالا بهینه نمی باشد.

اگر بعد از چک کردن تمام حالات قانون جدید ما درست باشد می توان ان را به knowledge base اضافه کرد و آن را برای موارد بعدی استفاده کرد.

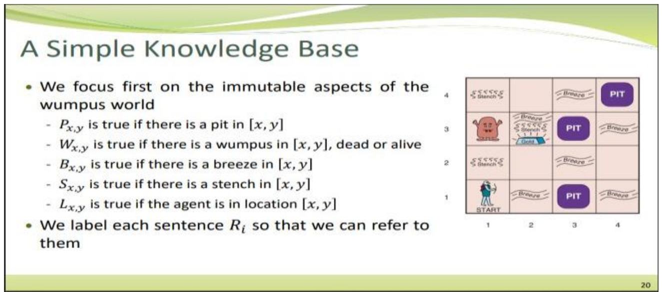
• Idea:
 – To test whether $\alpha \models \beta$, enumerate all models and check truth of α and β.
 – α entails β if no model exists in which α is true and β is false (i.e. $(\alpha \wedge \neg\beta)$ is unsatisfiable)
• Proof by Contradiction: $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.
• Model Checking:
 – Variables: One for each propositional symbol
– Domains: {true, false}
– Objective Function: $(\alpha \wedge \neg\beta)$

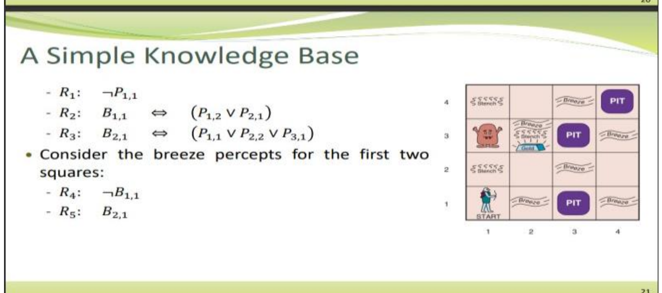
In order to solve such a problem algorithmically, both the model of the system and its specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a structure satisfies a given logical formula. This general concept applies to many kinds of logic and many kinds of structures. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure.

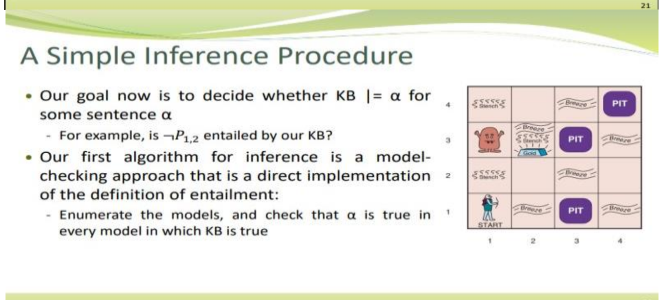
مثال :

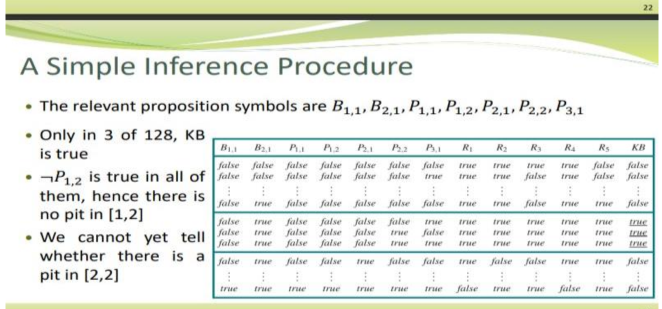
برای مثال این قسمت ابتدا از اسلاید ها کمک میگیریم

مانند مسئله ای که در آن یک غول داشتیم و میخواستیم ببینیم در خانه ی مورد نظر برای مثال چاه وجود دارد یا نه ؟
برای این کار تمامی حالات مختلف موجود بازی را در نظر میگیریم و با توجه به دیتاهایی که در knowledge base داریم بررسی میکنیم که فرض ما در نهایت enties می شود یا نه با توجه به شرایط و گزاره های موجود در این موقع تک تک حالات را بررسی می کنیم تا ببینیم با فرض موجود به تناقض میرسیم یا خیر و در نهایت به یک نتیجه درست خواهیم رسید.









$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	false	true	false	true	true	true	true	true	true
false	true	false	false	false	true	true	true	true	true	true	true	true
false	true	false	false	true	false	false	true	false	false	true	true	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	true	true	true	true	true	true	false	true	true	false	true	false

مثال دوم :

بلور در اثر خوردن به جسم سخت می کشند , لیوان شیشه ای داریم که در اثر افتادن می شکند.

لیوان شیشه ای از جنس بلور است : B

شکست بلور : SH

شکستن لیوان : L

نتیجه : R : L and SH ➔ B

در row ها که kβ مقدار یک دارد B هم یک می شود پس فرض درست بوده و اثبات می شود و به knowledge base اضافه میشود.

Kβ : SH , L ,R~ 1

B	L	SH	R	Kβ
1	1	1	1	**1**
1	1	0	1	**0**
1	0	1	1	**0**
1	0	0	1	**0**
0	1	1	0	**0**
0	1	0	1	**0**
0	0	1	1	**0**
0	0	0	1	**0**

روش دوم :

در روش Theorem – proving ما برای اثبات فرضیات خود از منطق ریاضی استفاده می کنیم و به جای چک کردن همه ی حالات موجود از چند عبارت منطقی استفاده می کنیم تا درستی فرض خود را چک کنیم .

می توان برای فرض های پیچیده تر از شکستن ان ها به فرض های کوچک تر و ساده تر و استفاده از قواعد مختلف به صورت ترکیبی تا فرض entiesشود.

Method 2: theorem-proving

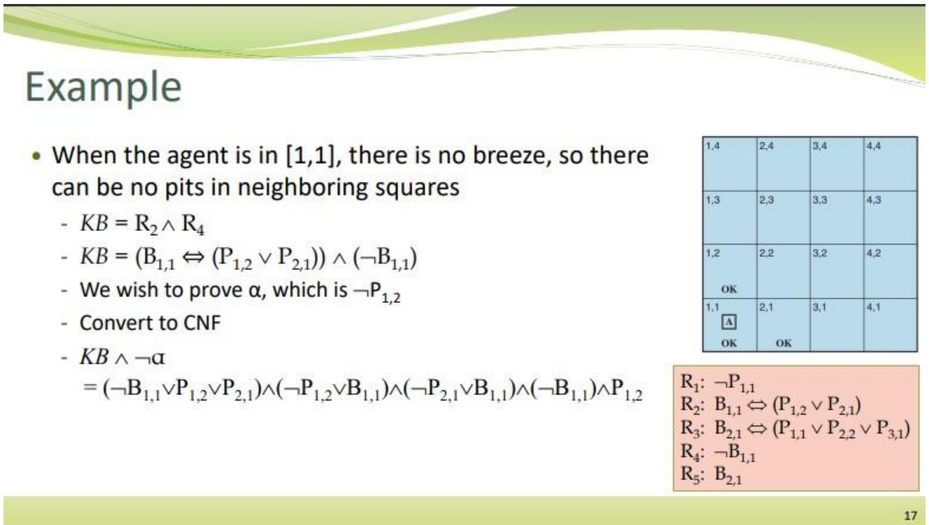
- Search for a sequence of proof steps (applications of inference rules) leading from α to β
- E.g., from P ∧ (P ⇒ Q), infer Q by Modus Ponens

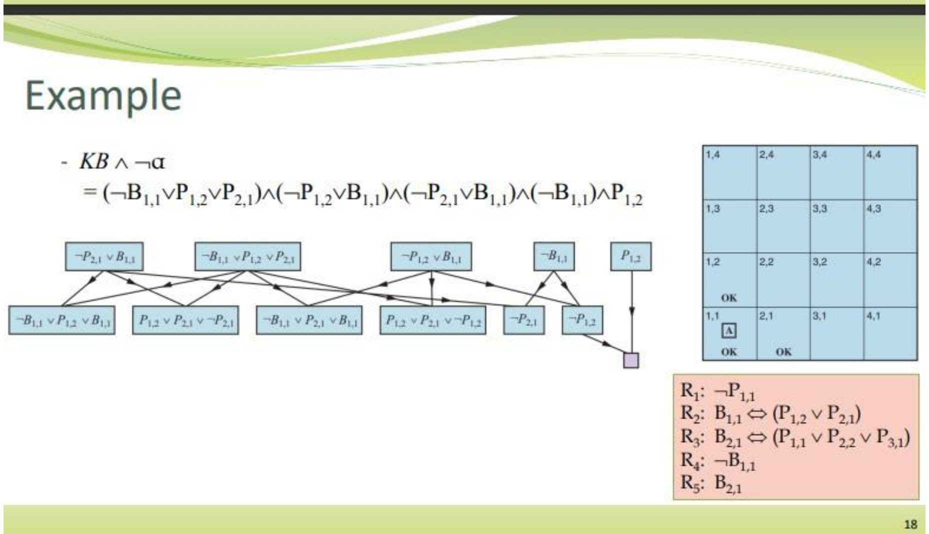
Reasoning by theorem proving is a weak method, compared to experts systems, because it does not make use of domain knowledge. This, on the other hand, may be a strength, if no domain heuristics are available (reasoning from first principles). Theorem proving is usually limited to sound reasoning.

Examples of inference rules

name	from	derive
modus ponens	p, p->q	q
modus tollens	p->q, ~q	~p
and elimination	p/\q	p
and introduction	p, q	p/\q
or introduction	p	p\/q
instantiation	for all X p(X)	p(a)
rename	for all X phi(X)	for all Y phi(Y)
exists-introduction	p(a)	exists X p(X)
substitution	phi(p)	phi(psi)
replacement	p->q	~p\/q
implication	assume p ... ,q	p->q
contradiction	**assume ~p ...,false**	**p**
resolution	**p\/phi, ~p\/psi**	**phi \/ psi**
(special case)	p, ~p\/psi	psi
(more special case)	p, ~p	false

حل مثال های قبلی با استفاده از روش دوم :





مثال دوم :

SH = 1 , L=1 ➔ L and SH =1

L and SH =1 , R =1 ➔ B =1

بخش عملی

بخش تحلیل نتایج کد ها :

- Reflex Agent

```
class TreeNode:
    def __init__(self, board, player, parent=[], action=[]):
        self.board = board
        self.player = player
        self.totalRollouts = 0.00
        self.totalCount = 0.00
        self.parent = parent
        self.action = action
        self.children = None
        self.ucb = -float('inf')

    def findNeighber(self):
        if self.children != None:
            return self.children
        else:
            neighbers = []
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == '_':
                        copyBoard = copy.deepcopy(self.board)
                        copyBoard[i][j] = self.player
                        neighbers.append(
                            TreeNode(copyBoard, opponent if self.player
== player else player, self, [i, j]))
            self.children = neighbers
            return self.children

    def calculateUCB(node):

        constant = 4
        if node.totalRollouts == 0:
            return float('inf')
        return (node.totalCount / node.totalRollouts) + constant *
math.sqrt(math.log(node.parent.totalRollouts) / node.totalRollouts)

    def isFinalState(self):
        return checkWin(self.board) or not isMovesLeft(self.board)
```

تحلیل بخش یک :

- در این سوال به پیاده سازی class Tree Node می پردازیم

در این قسمت ابتدا در قسمت __init__ متغیر هایی را تایین می کنیم که در طول برنامه از ان ها استفاده می کنیم سپس در قسمت findNeighbor پیدا می کنیم چه استیت هایی خانه خالی دارند که ان ها را انتخاب کنیم

- Calculate UCB / Final State

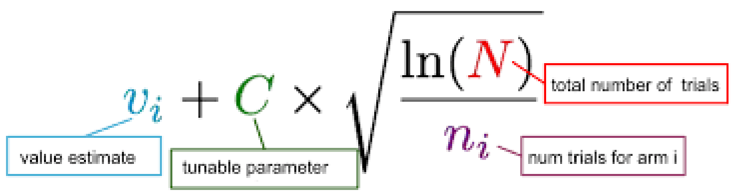
```
def calculateUCB(node):

        constant = 4
        if node.totalRollouts == 0:
            return float('inf')
        return (node.totalCount / node.totalRollouts) + constant *
math.sqrt(math.log(node.parent.totalRollouts) / node.totalRollouts)

    def isFinalState(self):
        return checkWin(self.board) or not isMovesLeft(self.board)
```

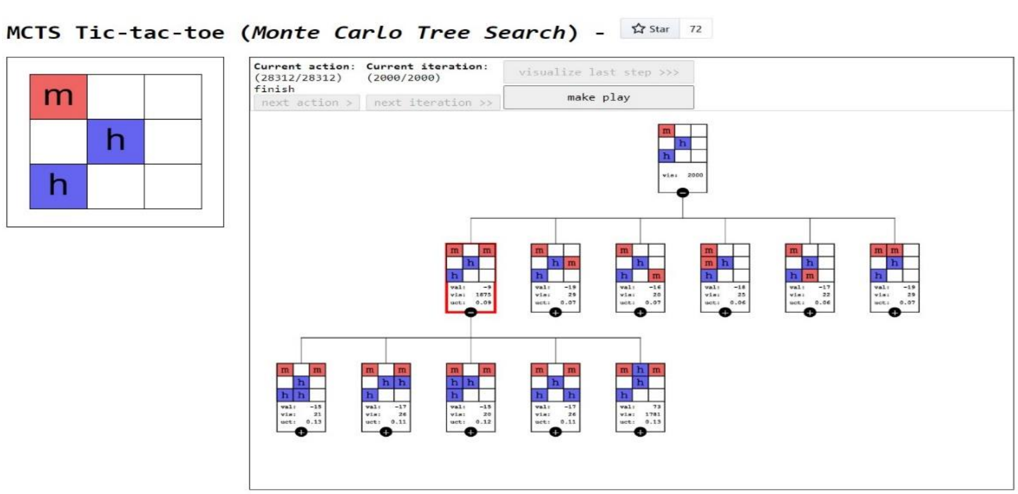
تحلیل بخش دو :

- در این سوال به پیاده سازی دو تابع کمکی Calculate UCB / Final State می پردازیم

در این قسمت می دانیم که درخت جست و جوی مونت کارلو با استفاده از فرمول مشخصی احتمال انتخاب شاخه ها با احتمال کم را بالا می برد .

و در تابع دیگر شرایط پایان یافتن استست ها را چک کردیم.



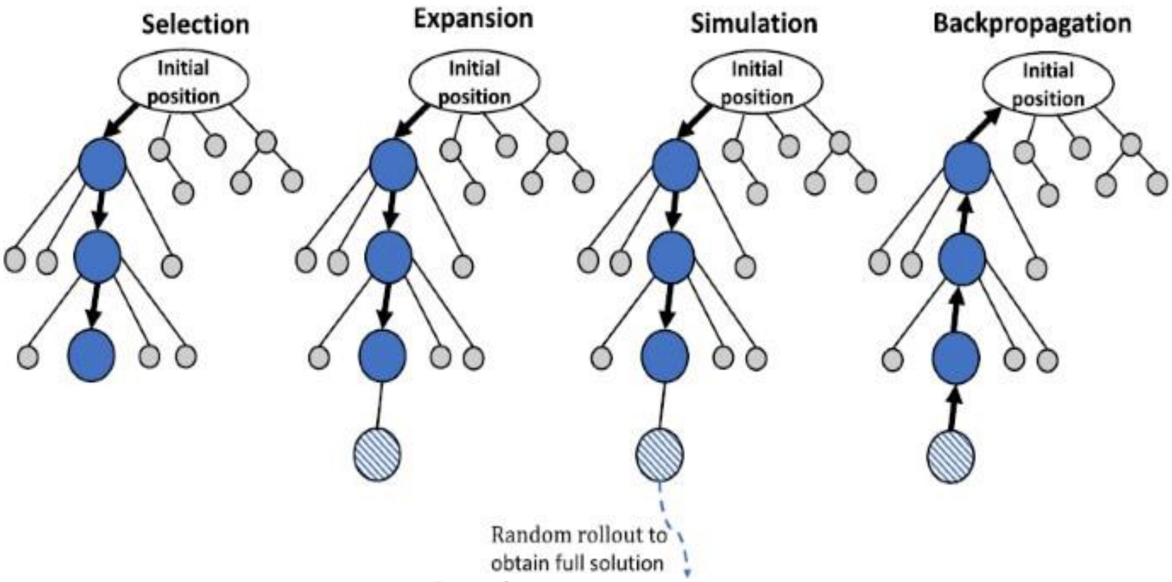


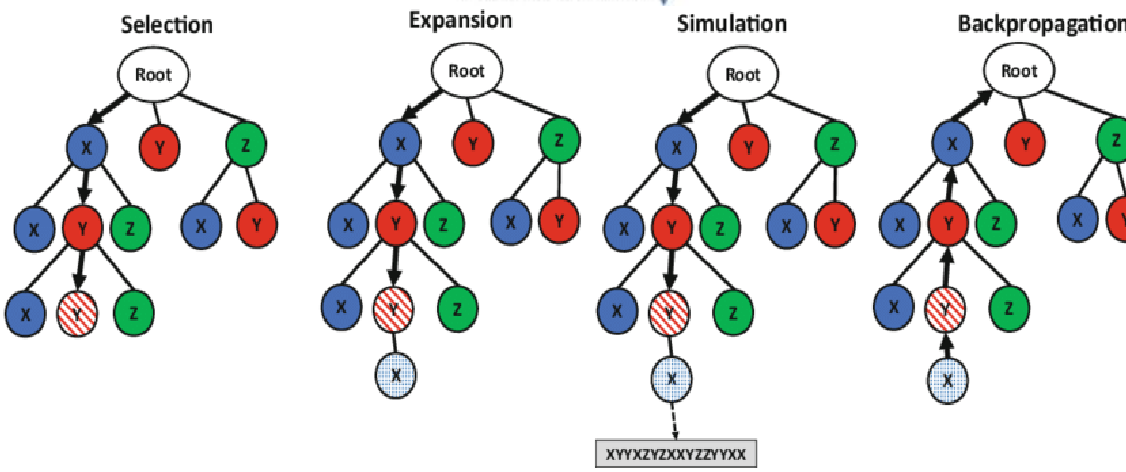
تحلیل بخش سه :

- در این سوال به پیاده سازی چهاربخش اصلی الگوریتم مونت کارلو می پردازیم

```python
def _selection_(node):
    if node.isFinalState():
        return node
    if node.totalRollouts == 0:
        return node
    maxScore = -float('inf')
    selected = None
    for child in node.findNeighber():

        score = child.calculateUCB()
        if score > maxScore:
            maxScore = score
            selected = child
    return _selection_(selected)


def _expansion_(node):
    neighbers = node.findNeighber()
    return choice(neighbers)


def _simulation_(node):
    board = copy.deepcopy(node.board)
    P = node.player
    while not checkWin(board) and isMovesLeft(board):
        i, j = findRandom(board)
        board[i][j] = P
        P = opponent if P == player else player
    return calculateScore(board, P)


def _backpropagation_(node, utility):
    node.totalRollouts = 1 + node.totalRollouts
    node.totalCount = utility + node.totalCount
    if node.parent:
        _backpropagation_(node.parent, utility)
```

چهار بخش اصلی این الگوریتم را مشاهده می کنیم :





3.1. Selection

In this initial phase, the algorithm starts with a root node and selects a child node such that it picks the node with maximum win rate. We also want to make sure that each node is given a fair chance.

The idea is to keep selecting optimal child nodes until we reach the leaf node of the tree. A good way to select such a child node is to use UCT (Upper Confidence Bound applied to trees) formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

In which

wi = number of wins after the i-th move

ni = number of simulations after the i-th move

c = exploration parameter (theoretically equal to √2)

t = total number of simulations for the parent node

The formula ensures that no state will be a victim of starvation and it also plays promising branches more often than their counterparts.

3.2. Expansion

When it can no longer apply UCT to find the successor node, it expands the game tree by appending all possible states from the leaf node.

3.3. Simulation

After Expansion, the algorithm picks a child node arbitrarily, and it simulates a randomized game from selected node until it reaches the resulting state of the game. If nodes are picked randomly or semi-randomly during the play out, it is called light play out. You can also opt for heavy play out by writing quality heuristics or evaluation functions.

3.4. Backpropagation

This is also known as an update phase. Once the algorithm reaches the end of the game, it evaluates the state to figure out which player has won. It traverses upwards to the root and increments visit score for all visited nodes. It also updates win score for each node if the player for that position has won the playout.

MCTS keeps repeating these four phases until some fixed number of iterations or some fixed amount of time.

In this approach, we estimate winning score for each node based on random moves. So higher the number of iterations, more reliable the estimate becomes. The algorithm estimates will be less accurate at the start of a search and keep improving after sufficient amount of time. Again it solely depends on the type of the problem.

منبع : https://www.baeldung.com/java-monte-carlo-tree-search

منبع : https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

انتخاب: از ریشه R شروع کنید و گره های فرزند متوالی را انتخاب کنید تا به گره برگ L برسید. ریشه وضعیت فعلی بازی است و برگ هر گره ای است که فرزند بالقوه ای دارد که هنوز هیچ شبیه سازی (بازی) از آن آغاز نشده است.
بخش زیر در مورد روشی برای انتخاب سوگیری گره‌های فرزند توضیح می‌دهد که به درخت بازی اجازه می‌دهد به سمت امیدوارکننده‌ترین حرکت‌ها گسترش یابد، که جوهره جستجوی درخت مونت کارلو است .بسط: مگر اینکه L بازی را قاطعانه به پایان برساند (به عنوان مثال برد/باخت/تساوی) برای هر یک از بازیکنان، یک (یا چند) گره فرزند ایجاد کنید و گره C را از یکی از آنها انتخاب کنید. گره های فرزند هر حرکت معتبری از موقعیت بازی تعریف شده توسط L هستند .
شبیه سازی: یک پخش تصادفی را از گره C کامل کنید. این مرحله گاهی اوقات پخش یا پخش نیز نامیده می شود. یک بازی ممکن است به سادگی انتخاب حرکات تصادفی یکنواخت باشد تا زمانی که بازی قطعی شود (مثلاً در شطرنج، بازی برنده، باخت یا مساوی می شود) .پس انتشار: از نتیجه پخش برای به روز رسانی اطلاعات در گره های مسیر C به R استفاده کنید.

تحلیل بخش چهار findBestMove

- در این سوال به پیاده سازی

```
ef calculateScore(board, turn):
    if checkWin(board):
        if turn == player:
            return 1
        else:
            return -1
    else:
        return 0.75

def findBestMove(board):

    root = TreeNode(board, opponent)
    for i in range(695):
        node = _selection_(root)
        if not node.isFinalState():
            node = _expansion_(node)
        utility = _simulation_(node)
        _backpropagation_(node, utility)
    result = max(root.findNeighber(),
                 key=lambda s: s.totalCount / s.totalRollouts).action
    return result


```

در تابع calculateScore با گرفتن دو پارامتر و بازگشت مقدار های مشخص برای برد و باخت و مساوی برمیگردانیم.
و در قسمت بعد با استفاده از چهار قسمت مونت کارلو تابع findBestMove را هوشمند کردیم و از حالت رندم بودن دراوردیم .

- Conclusion

۱ با استفاده از فایل q2.py و تست کردن برنامه در مجموع امتیاز ۵/۵ دریافت می شود.
۲ الگوریتم اولیه ی ما به طور کلی با شکست مواجه می شوند چون رندم است اما در این الگوریتم در حالت هایی بازی می کند که هوشمند است و حالات برد را افزایش داد.

عکس از فضای گرافیکی و تست کردن :

با اجرای دستورات

PS C:\Users\user\Desktop > python q2.py می توانیم محیط گرافیکی بازی را ببینیم.