بخش 0) فهمیدن نحوه ی کار عامل ReflexAgent

هر دور که تابع getAction برای این عامل صدا زده میشود ، به ازای همه ی action های ممکن ، تابع evaluationFunction را فراخوانی میکند و در نهایت آن action ای که evaluationFunction آن مقدار بزرگتری داشته را انتخاب میکند.

در evaluationFunction محاسبه را به این شکل انجام میدهد که امتیاز را با انجام دادن آن action برمیگرداند. در واقع با این evaluationFunction عامل هیچ توجهی به state فعلی ندارد (مثلا شاید اگر به چپ برود غذا بخورد و در getScore امتیاز این حرکت بیشتر باشد اما یک روح هم همانجا باشد و درجا بسوزد (و عامل این روح را در نظر نمیگیرد))

```
successorGameState = currentGameState.generatePacmanSuccessor(action)
newPos = successorGameState.getPacmanPosition()
newFood = successorGameState.getFood()
newGhostStates = successorGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
result = 0
foodDistances = [util.manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()]
if len(foodDistances) > 0:
    result += float(9 / min(foodDistances))
index = 0
scared_indexes = [i for i in newScaredTimes if i > 1]
minimum_scared_ghost_distance = 10000
for ghost in newGhostStates:
    distance = abs(newPos[0] - ghost.getPosition()[0]) + abs(newPos[1] - ghost.getPosition()[1])
    if index in scared_indexes and distance < minimum_scared_ghost_distance:
        minimum_scared_ghost_distance = distance
    elif distance > 1:
        if distance > 3:
            result += 3
        else:
            result += distance
    elif distance == 1:
        return -2000
    index += 1
if len(scared_indexes) > 0:
    result += float(1 / minimum_scared_ghost_distance)
return result + successorGameState.getScore()
```

بخش1) بهبود عملکرد ReflexAgent

حال باید با تغییر evaluationFunction این عامل ، عملکرد آنرا بهبود بدهیم.

یک result برای مقدار evaluationFunction مشخص کرده و به آن مقدار 0 میدهیم.

میدانیم پکمن باید به سمت غذا برود و رفتن به سمت غذای نزدیکتر اولویت دارد. برای همین کمترین فاصله تا غذا را در نظر میگیریم و معکوس آنرا با result جمع میکنیم. این به این معناست که هر چه فاصله مینیمم کمتر باشد score ما بیشتر میشود. حالا چرا به جای اضافه کردن معکوس مینیمم فاصله ، مستقیما خود فاصله را کم نکردیم؟ (با توجه به اینکه فاصله مینیمم را کم میکنیم نزدیکتر شدن به مینیمم score را خب بیشتر میکند و منطقی به نظر میرسد) . نکته اینجاست فرض کنید به یک قدمی یک غذا رسیده ایم. در حال حاضر مینیمم فاصله تا غذا 1 است. میدانیم اگر این غذا را بخوریم ، نزدیکترین غذای بعدی در فاصله 10 است. خوردن غذا هم یک امتیاز دارد . در اینصورت عامل ما براورد میکند، خوردن غذا 1 واحد اضافه میکنه و 10 واحد کم. نخوردن غذا و موندن در فاصله 1 با غذا فقط 1 واحد کم میکنه پس به صرفه است که کنار غذا توقف کنم. برای جلوگیری از این اتفاق مینیمم فاصله/1 را اضافه میکنیم که وقتی به یک غذا نزدیک شدیم بدانیم خوردن آن مقدار بیشتری به result اضافه میکند تا نخوردن و ایستادن کنارش (اگر کنار غذا باشیم خوردنش 1 واحد امتیاز دارد + 0.1 هم برای غذای بعدی به score اضافه میکند اما نخوردن غذا صرفا 1 واحد به score اضافه میکند(1/1))

حال بعد از در نظر گرفتن غذا ها نوبت به در نظر گرفتن موقعیت روح هاست.

ابتدا ایندکس همه ی روح هایی که ترسیده هستند و تایم ترسیده بودنشن بیشتر یا مساوی 2 است (اگر 1 باشد ممکن است با نزدیک شدن به آن بسوزیم مثلا در ثانیه بعدی)را به دست میاوریم و ضمنا متغییری برای نزدیکترین روح ترسیده نیز تعیین میکنیم.به ازای همه ی روح ها باید یکسری شرط را بررسی کنیم. شرط اول اینست که اگر روح ترسیده بود چک کنیم نزدیکترین روح ترسیده است یا نه. اخرین شرط اینست که اگر روح در یک قدمی ما بود result را به شدت کم کنیم که این action احتمال انتخابش خیلی خیلی کم شود. اگر فاصله روح با ما بیشتر بودبه اندازه فاصله اش به result اضافه میکنیم چون هرچه روح دورتر باشد بهتر است. حال نکته اینجاست در اینصورت پکمن ممکن است در یک گوشه توقف کند چون بیشترین فاصله از روح ها را دارد و همه چیز عالی است. برای جلوگیری از این حالات تعیین میکنیم اگر فاصله ی روح تا پکمن از 3 واحد بیشتر بود result را فقط 3 واحد اضافه کنیم نه به اندازه فاصله ی روح تا پکمن.ما از فاصله نزدیکترین روح ترسیده استفاده ای نکردیم. در اخر به اندازه معکوس فاصله نزدیکترین روح ترسیده به result اضافه میکنیم(در واقع به روح ترسیده عین غذا نگاه میکنیم.)

در نهایت result را با score جمع میکنیم چون گذر زمان نیز از امتیاز کم میکند و ... و این موارد نیز باید در نظر گرفته شوند.

تست تابعمان نیز نشان میدهد به طور میانگین بالا 1000 امتیاز گرفته و در هر دور بازی برنده شده.

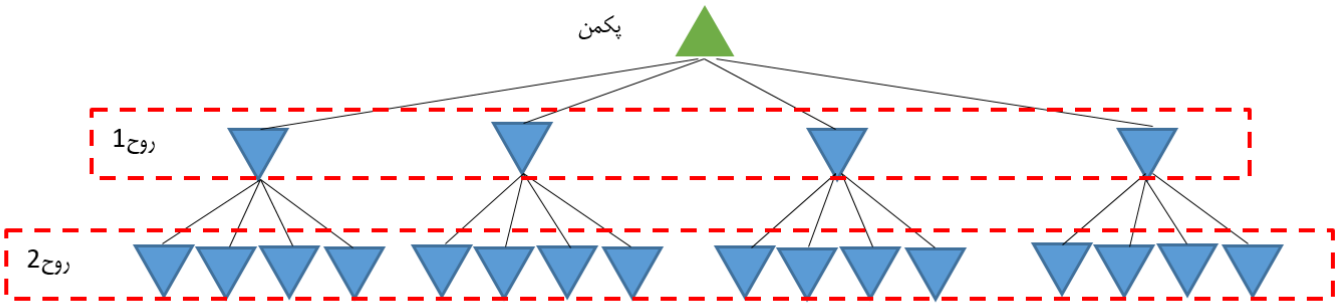
```
Pacman emerges victorious! Score: 1234
Pacman emerges victorious! Score: 1231
Pacman emerges victorious! Score: 1237
Pacman emerges victorious! Score: 1236
Pacman emerges victorious! Score: 1238
Pacman emerges victorious! Score: 1238
Pacman emerges victorious! Score: 1241
Pacman emerges victorious! Score: 1236
Pacman emerges victorious! Score: 1252
Pacman emerges victorious! Score: 1247
```

بخش 2) مینیماکس

ما دو نوع agent داریم ، روح و پکمن. پکمن هدفش اینست که همه غذا ها را بخورد و بدون برخورد با روح ها بیشترین امتیاز را بگیرد و روح میخواهد پکمن را بخورد. حال یک استراتژی برای پکمن اینست که در هر استیتی که قرار دارد، تا depth بار، فرض کند روح ها حرکتی را انجام میدهند که بیشترین ضرر را برایش دارد، یعنی فرض میکند روح ها agent های با همین استراتژی خودش هستند(که به دنبال بیشترین ضرر طرف مقابلند). بعد بین حالاتی که رخ میدهد آن حالتی را انتخاب میکند که بیشترین مقدار امتیاز را کسب کند.

همین توضیح بالا ما را به مفهوم درخت minimax میرساند. دو نوع node داریم که maximizer و minimizer هستند و همانطور که از اسمشان مشخص است، maximizer ماکسیمم را از گره های فرزندش انتخاب میکند و minimizer مینیمم را.

Minimax را به نحوی پیاده سازی میکنیم که به ازای هر تعداد روح درست کار کند. در نظر داریم که agent ها به ترتیب ایندکسشان عمل میکنند و پکمن ایندکس 0 دارد. درخت minimax ما برای 2 روح به شکل زیر میشود حال برای تعداد بیشتر روح همین روند را کافیست تکرار کنیم.



همانطور که در شکل میبینید، به ازای تمام حرکات ممکن برای یک عامل ، حالاتی که در پی آن رخ میدهد را باید ارزیابی کنیم، در agent های minimizer از بین ان حالات مینیمم و در maximizer ها ماکسیمم را انتخاب کنیم. باتوجه به چند روحی بودن بازی و ترتیب بازی کردنشان، در درخت روی یک مسیر تا ترمینال، k مینیمایزر پشت سرهم داریم و یک ماکسیمایزر که همان پکمن است بعد و قبل آنها.(توجه شود درخت بالا ادامه دارد و رسم نشده)

در پیاده سازی باید در نظر داشته باشیم عمق هم اهمیت دارد و از درختمان تا عمق n را بررسی کنیم. شاید تا عمق n اصلا به ترمینال ها نرسیده باشیم بنابراین به evaluating function احتیاج داریم تا براوردی از وضعیت استیت به ما بدهد. ضمنا بررسی تا عمق n یعنی هر عامل n حرکت انجام داده باشد نه اینکه عمق درخت n باشد.

طبق توضیحات minimax را پیاده سازی میکنیم. پیاده سازی دو بخش دارد:

```
ghosts_num = gameState.getNumAgents()

def maximumValue(gameState, depth):
    max_value = float('-inf')
    depth = depth - 1
    legal_actions = gameState.getLegalActions(0)
    if depth == 0 or gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState), 'stop'
    for action in legal_actions:
        value = minimumValue(gameState.generateSuccessor(0, action), depth, 1)
        if value > max_value:
            max_value = value
            next_action = action
    return max_value, next_action
```

1) تابعی برای گره های maximizer تعریف میکنیم (همان پکمن). این تابع عمق و gameState را به عنوان ورودی میگیرد و ماکسیمم و action برای رسیدن به آن را برمیگرداند.

در تابع ابتدا از عمق یک واحد کم میکنیم. سپس کل حرکات ممکن در این استیت برای عامل مکسیمایزر(همان پکمن که agent با ایندکس 0 است) را در لیستی نگه میداریم. با انجام دادن اکشن، بقیه عامل ها یعنی روح ها به ترتیب کارشان را انجام میدهند و گفتیم آنها مینیمایزرند پس ما به ازای هر اکشن پکمن مقدار نهایی گره ی بعدی (حالا یا با استفاده از evaluating function یا با استفاده از مقدار واقعی ترمینال ها) را به دست می اوریم و ماکسیمم آنها را انتخاب میکنیم. سپس اکشنی که برایمان بهینه است همان اکشنی است که مارا به ماکسیمم میرساند.

max_value را بیشترین مقدار ممکن میدهیم تا بعدا که وارد حلقه ی انتخاب ماکسیمم میشویم، عددش عوض شود. برای به دست اوردن ماکسیمم از تابع از پیش تعریف شده ی max استفاده میکنیم

برای به دست اوردن نتیجه ی اکشن های روح ها تابع minimumValue را به ازای استیت بعد از هر اکشنمان فراخوانی میکنیم ضمنا با تابع generateSuccessor وضعیت بعد از انجام اکشن را به دست می اوریم.

دقت شود که در صورتی که عمق صفر شود، یا برنده شویم یا مشخص شود که بازنده ایم بدون بررسی فرزند ها مستقیما مقدار evaluationFunction را برمیگردانیم و کار تمام است.

2)تابعی برای گره های minimizer به اسم minimumValue تعریف میکنیم. این تابع عمق و ایندکس عامل و gameState را به عنوان ورودی میگیرد و مینیمم را به عنوان خروجی میدهد. دقت شود دیگر اکشنی که ما را به مینیمم هدایت میکند را ازش نمیگیریم چون این عوامل همان روح ها هستند و حرکت آنها برای ما اهمیتی ندارد چون ما به آنها برنامه ی حرکت قرار نیست بدهیم.

```
def minimumValue(gameState, depth, ghost_index):
    min_value = float('inf')
    legal_actions = gameState.getLegalActions(ghost_index)
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    for action in legal_actions:
        if ghost_index < (ghosts_num - 1):
            min_value = min(min_value,
                            minimumValue(gameState.generateSuccessor(ghost_index, action), depth, ghost_index+1))
        else:
            min_value = min(min_value, maximumValue(gameState.generateSuccessor(ghost_index, action), depth)[0])
    return min_value
```

Min_value را بیشترین مقدار ممکن میدهیم تا بعدا که وارد حلقه ی انتخاب مینیمم میشویم، عددش عوض شود. برای به دست اوردن مینیمم از تابع از پیش تعریف شده ی min استفاده میکنیم.

در این تابع هم لیست اکشن های مجاز را به دست می آوریم. حال به ازای هر اکشن باید بررسی کنیم که برای انتخاب فرزند چه تابعی را فراخوانی کنیم .اگر ایندکس روح متعلق به آخرین روح باشد یعنی عامل بعدیی که حرکت میکند پکمن است که maximizer است پس مینیمم را باید بین مقادیری که از maximumValue برای پکمن و state بعدی حاصل میشود انتخاب کنیم. اگر ایندکس متعلق به روح های غیر از اخرین روح باشد مینیمم فرزندان را با تابع minimumValue را برای حرکت بقیه روح ها بدست می اوریم.

ضمنا باز هم اگر برنده یا بازنده بودن مشخص شد دیگر نیاز به فراخوانی های متعدد نیست و کافیست مقدار evaluatingFunction برگردانده شود.

در نهایت عبارت زیر را برمیگردانیم

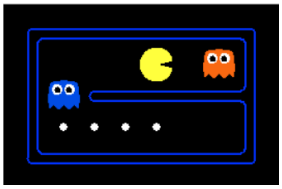
maximumValue(gameState, self.depth + 1)[1]

maximumValue به این علت فراخوانی میشود که ابتدا پکمن حرکتش را انجام میدهد و بعد بقیه ی روح ها. عمق با یک جمع شده چون در تابع mazimumValue ابتدا یک واحد از عمق کم میشود و بعد بررسی ها صورت میگیرد . ضمنا گفتیم خروجی این تابع هم ماکسیمم هست و هم حرکتی که منجر به آن میشود. درون تابع getAction این minimax را پیاده سازی کرده ایم پس خروجی صرفا یک action است بنابراین مولفه ی دوم خروجی تابع maximumValue را برمیگردانیم.

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState):

        ghosts_num = gameState.getNumAgents()

        def maximumValue(gameState, depth):...

        def minimumValue(gameState, depth, ghost_index):...

        return maximumValue(gameState, self.depth + 1)[1]
```

خروجی autograder که نشان از درستی عملکرد minimax برای maze های با چند روح دارد.

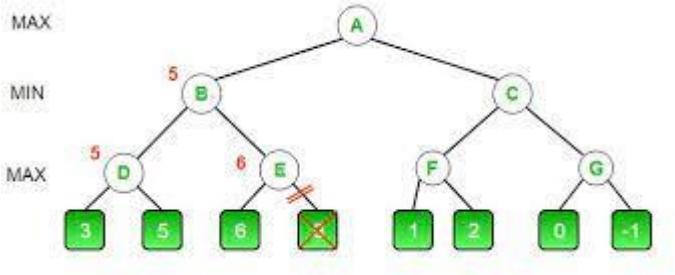
```
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q2\8-pacman-game.test

### Question q2: 5/5 ###
```



سوال) در نقشه رو به رو میبینیم پکمن به راست میرود تا زودتر بمیرد. این به این معناست که پکمن تشخیص داده قطعا خواهد مرد. چرا؟
چون فرض کرده روح ها به بهترین نحو ممکن عمل میکنند یعنی روح آبی قطعا بالا میاید. اگر روح آبی بالا بیاید چکمن از دو سمت توسط روح هایی که به سمتش حرکت میکنن احاطه شده بنابراین چون گذر زمان هزینه دارد و ما میخواهیم هزینه ها کمتر باشد خودش را به سمت روح نزدیکتر میبرد که زودتر بمیرد.

بخش 3) هرس الفا بتا:

تا الان که الگوریتم مینیماکس را پیاده سازی کردیم، برای یافتن جواب کل گره ها را بررسی میکرد که گاها نیاز به بررسی وجود نداشت. به مثال توجه کنید.



بعد از مشخص کردن مقدار گره ی D که برابر با 5 شده میدانیم گره B بین D , E مینیمم را انتخاب میکند یعنی یا 5 (که گره B است) یا گره E در صورتی که مقدارش کمتر از 5 شود. پس مقدار گره B کمتر یا مساوی 5 است. حال به بررسی گره E میپردازیم. بعد از بررسی فرزندش که 6 است چون میدانیم گره E یک ماکسیمایزر است، مطمین هستیم مقدارش یا 6 میشود یا بیشتر از 6. قبلا گفتیم B یا 5 است یا کوچکتر از 5 پس ابدا متغییر E را برای مینیمم بودن انتخاب نمیکند و نیاز نیست به بررسی بقیه فرزندان E بپردازیم.

الگوریتم هرس الفا بتا مانند مثال بالا از بررسی گره هایی که تاثیری روی جواب ندارند جلوگیری میکند. آلفا نشان دهنده ی حداکثر کران پایین و بتا نشان دهنده حداکثر کران بالا برای بهترین گزینه در مسیر ریشه است. MAXIMIZER ها مقدار الفا و MINIMIZER ها مقدار بتا را تغییر میدهند.

طبق الگوریتم زیر کد مینیماکس را با کمی تغییر ارتقا داده و هرس الفا بتا را به آن اضافه میکنیم.

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

الفا و بتا فقط به گره های فرزند داده میشوند و هنگام BACKTRACK کردن فقط مقدار خود گره ها را به والدشان میدهیم.

در تابع گره ماکسیمایزر ، الفا و بتا را هم به عنوان ورودی اضافه کرده ایم. طبق الگوریتم اگر مقدار فرزندان از ماکسیمم ممکن بیشتر میشد هرس انجام شده و دیگر فرزندان آن فرزند بررسی نمیشوند.

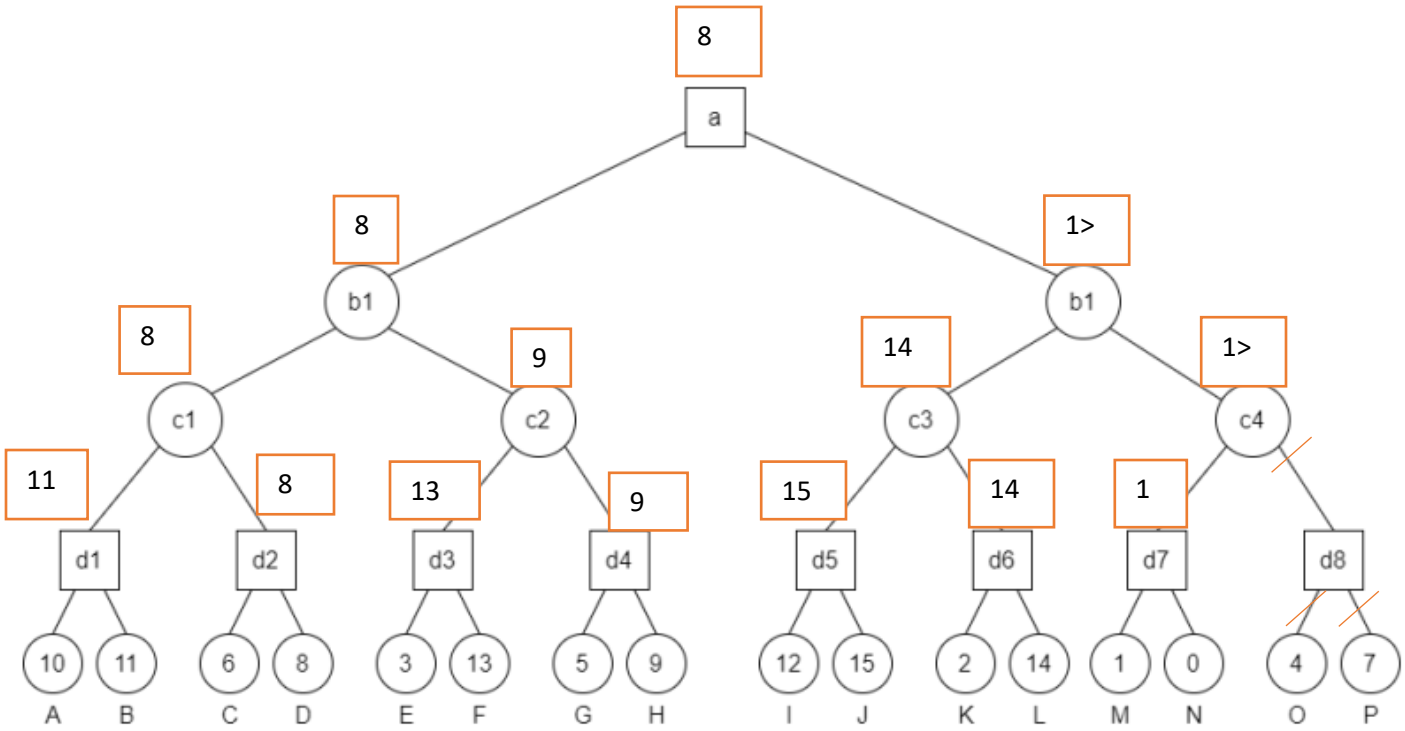
این بخش در کادر قرمز اضافه شده است.

```
def maximumValue(gameState, depth, alpha, beta):
    max_value = float('-inf')
    depth = depth - 1
    legal_actions = gameState.getLegalActions(0)
    if depth == 0 or gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState), 'stop'
    for action in legal_actions:
        value = minimumValue(gameState.generateSuccessor(0, action), depth, 1, alpha, beta)
        if value > max_value:
            max_value = value
            next_action = action
        if value> beta: return value,next_action
        alpha = max(alpha, max_value)

    return max_value, next_action
```

در تابع مینیمایزر نیز الفا و بتا به عنوان ورودی داده شده و طبق الگوریتم اگر مقدار فرزندان از مینیمم ممکن کمتر میشد آن زیر درخت هرس میشود و فرزندانش را بررسی نمیکنیم.

در کادر قرمز این بخش را میبینیم.

```
def minimumValue(gameState, depth, ghost_index_, alpha, beta):
    min_value = float('inf')
    legal_actions = gameState.getLegalActions(ghost_index)
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    for action in legal_actions:
        if ghost_index < (ghosts_num - 1):
            min_value = min(min_value,
                            minimumValue(gameState.generateSuccessor(ghost_index, action), depth,
                                         ghost_index + 1_, alpha, beta))
        else:
            min_value = min(min_value, maximumValue(gameState.generateSuccessor(ghost_index, action), depth, alpha, beta)[0])
        if min_value < alpha: return min_value
        beta = min(beta, min_value)
    return min_value
```

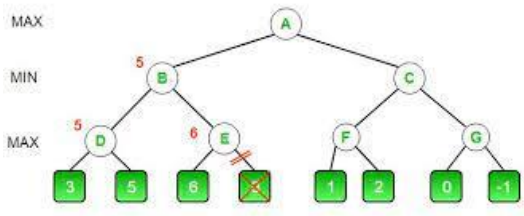


بعد از بررسی گره ی d7 میدانیم که c4 کمتر یا مساوی یک است. به طور مشابه میدانیم b1 نیز چون مینیمایزر است کمتر یا مساوی یک خواهد بود در نتیجه قطعا b1 توسط ریشه که ماکسیمایزر است انتخاب نخواهد شد چون گزینه 8 از b1 بزرگتر خواهد بود پس دیگر نیاز به بررسی d8 و فرزندانش(o,p) نداریم و آنها هرس میشوند.

طبق نتیجه ی درخت، بهتر است پکمن به چپ حرکت کند و وارد b1 شود تا در بدترین حالت امتیاز بیشتری کسب کند.

سوال)

درخت الفا بتا در ریشه مقدار متفاوتی نمیتواند تولید کند اما در گره های میانی میتواند. همانطور که در مثال بالا دیدید در برخی گره ها نوشتیم بزرگتر یا مساوی عدد k . در این حالت ما مقدار دقیق گره میانی را نمیدانیم اما دلیل آن اینست که آن گره در جواب والدش تاثیری ندارد و انتخاب نمیشود و در نهایت ریشه را نیز تحت تاثیر قرار نخواهد داد.



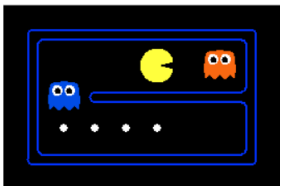
در همین مثال میبینیم مقدار گره E 6 شده اما در واقعیت باید 9 میبود اما این واقعی نبودن مقادیر گره های میانی تاثیری در ریشه ندارد چون مطمین هستیم گره E توسط والدش انتخاب نخواهد شد.

بخش 4- در مینیماکس و هرس الفا بتا ما همواره تصور میکردیم حریف به اندازه ما هوشمندانه عمل میکند و بهترین انتخاب های خودش که برای ما بدترین هستند را انجام میدهد بنابراین مثلا در برخی شرایط سعی میکردیم زودتر بمیریم چون میدانستیم اگر حریف هوشمندانه عمل کند در هر صورت میبازیم پس بهتر است کمتر هزینه زندگی بدهیم.

حال در این بخش تغییری ایجاد میکنیم. فرض میکنیم گره های مینیمایزر گره های شانس هستند و بین حرکات مجازشان به صورت احتمالی یکی را انتخاب میکنند و این احتمال ها بین حرکات یکسان است. در این صورت مقدار گره ی شانس برابر با جمع مقدار های فرزندانش ضربدر احتمالشان میشود (یعنی مقدار گره ی شانس در واقعیت ممکن است هیچ وقت به دست نیاید). گره های ماکسیمایزر تغییری نمیکنند و فرض میکنیم خودمان همیشه بهترین حالت را انتخاب میکنیم.

```
def chanceValue(gameState, depth, ghost_index):
    sum =0
    legal_actions = gameState.getLegalActions(ghost_index)
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    for action in legal_actions:
        if ghost_index < (ghosts_num - 1):
            sum = sum + chanceValue(gameState.generateSuccessor(ghost_index, action), depth, ghost_index+1)
        else:
            sum = sum + maximumValue(gameState.generateSuccessor(ghost_index, action), depth)[0]
    return float(sum/len(legal_actions))

return maximumValue(gameState, self.depth + 1)[1]
```

به جای تابع minimumValue که برای گره های مینیمایزر بود از تابع chanceValue به عنوان مقدار گره ی شانس استفاده میکنیم. گفتیم احتمال وقوع حرکات مختلف را یکسان در نظر میگیریم پس کافیست بین مقادیر فرزندان میانگین بگیریم. ابتدا مقادیر فرزندان در sum جمع میشود سپس بر تعداد حرکات ممکنی که داشتیم(تعداد فرزندان) تقسیم میشود. توجه کنید اگر احتمال ها برابر نبود یک متغییر value در نظر میگرفتیم و هربار مقدار فرزند را ضربدر احتمال انتخابش میکردیم و به value اضافه میکردیم و درنهایت value را برمیگرداندیم.



سوال)

در maze رو به رو دو الگوریتم را بررسی میکنیم (اگر حریف هوشمندانه عمل کند همواره ما میبازیم)

در مینیماکس با هرس الفا بتا نتایج زیر حاصل میشود.

```
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

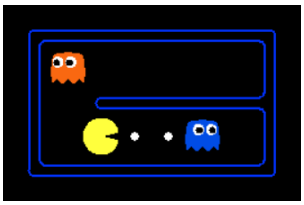
میبینیم که در هر دور بازی پکمن سعی کرده زودترین مرگ را داشته باشد و در همه ی بازی ها باخته (و مقدار این باخت هم همیشه یکسان است)

حال در expectimax نتایج زیر حاصل میشود.

```
Average Score: 325.2
Scores:        532.0, 532.0, 532.0, 532.0, 532.0, -502.0, 532.0, 532.0, -502.0, 532.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Win, Win, Win, Loss, Win, Win, Loss, Win
```

همانطور که در نتایج مشخص است از 10 بار 8 بار پکمن برنده شده است. البته چون احتمال دخیل است باید ازمایش در تعداد بالا انجام شود. اگر در نمونه های زیادی با expectimax بازی کنیم حدودا در 50 درصد مواقع پکمن برنده خواهد شد.

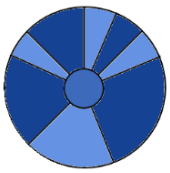
حال چرا در expectmax ممکن است پکمن برنده شود؟ به مثال زیر توجه کنید.



چون روح ها به طور احتمالی حرکت میکنند، روح ابی به جای اینکه با بالا رفتن پکمن را به دام بیندازد به پایین حرکت کرده و پیش میرود اما پکمن حرکات احتمالی نمیکند بلکه با دیدن اینکه روح آبی پایین رفته و درختی expectimax تصمیم میگیرد به سمت دات ها برود.

سوال) الگوریتم رولت ویل

کروموزوم ها تابعی برای fitness دارند. هر کروموزومی که f بزرگتری داشته باشد برای انتخاب بهتر است. اما نمیخواهیم n کروموزوم برتر را انتخاب کنیم بلکه میخواهیم همه ی کروموزوم ها شانس بودن در دسته ی انتخابی را داشته باشند اما انهایی که f بزرگتری دارند شانس بیشتری داشته باشند. برای اینکار برای هر کروموزوم یک احتمال در نظر میگیریم. جمع این احتمال ها باید یک شود. احتمال انتخاب کروموزوم i ام را $\frac{f_i}{\sum f}$ در نظر میگیریم. زیگما که جمع f های همه ی کروموزوم هاست ثابت است اما صورت کسر در کروموزوم هایی که f بزرگتری دارند بزرگتر است پس احتمال انتخباشان بیشتر خواهد بود. این مدل انتخاب را به چرخاندن چرخی به شکل زیر تشبیه میکنند که بخش های آن هم اندازه نیستند و احتمال برخورد تیر(چاقو) ی ما با برخی ناحیه ها بیشتر است.



از این الگوریتم در پکمن خودمان نیز میتوانیم کمک بگیریم. در expectimax برای روح ها احتمال همه ی حرکات را یکسان در نظر گرفتیم اما میتوانیم فرض کنیم روح ها اکثرا تلاش میکنند به ضرر ما عمل کنند و برخی اوقات اشتباه میکنند. با این فرض به حالاتی که امتیاز ما را کم کنند احتمال بیشتری میدهیم و به حالاتی که امتیاز ما را زیاد کنند امتیاز کمتری میدهیم (میتوانیم احتمال انتخاب هر اکشن را بر اساس evaluation حالت بعدش حساب کنیم $\frac{\max(e)+1-e_i}{n*\max(e)+n-\sum e}$ یا $\frac{\sum e-e_i}{(n-1)*\sum e}$ $= \frac{\sum e-e_i}{n*\sum e-\sum e}$ توابعی برای احتمال انتخاب فرزند i ام از n فرزند بر اساس تابع e است که فرزندانی با e کمتر احتمال انتخاب بیشتری دارند و این شبیه کار روح ها میتواند باشد)

برای گسترش 2حالت به صورت همزمان، ابتدا 2 حالت را انتخاب میکنیم و بعد برای حالت های انتخابی نیز 2 حالت را انتخاب میکنیم و انقدر پیش میرویم تا به ترمینال ها برسیم، در آخر بین ترمینال ها انتخاب میکنیم. خوبی این حالت اینست که اگر انتخاب اولمان نتیجه خوبی نداد انتخاب های دیگرمان هستند که احتمالا نتایج خوبی دارند.

بخش 5-

در بخش 5 evaluate function بخش 1 را بهبود میدهیم. بنابراین صرفا تغییرات نسبت به بخش 1 توضیح داده میشود.

```
newPos = currentGameState.getPacmanPosition()
newFood = currentGameState.getFood()
newGhostStates = currentGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
capsules = currentGameState.getCapsules()
result = 0
foodDistances = [util.manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()]
if len(foodDistances) > 0 : result -=1/100* (max(foodDistances) + min(foodDistances))
index = 0
scared_indexes = [i for i in newScaredTimes if i > 0]
minimum_scared_ghost_distance = 10000
for ghost in newGhostStates:
    distance = abs(newPos[0] - ghost.getPosition()[0]) + abs(newPos[1] - ghost.getPosition()[1])
    if index in scared_indexes and distance < minimum_scared_ghost_distance:
        minimum_scared_ghost_distance = distance
    elif distance > 1:
        if distance > 3:
            result += 3
        else:
            result += distance
    elif distance == 1:
        return -2000
    index += 1
if len(scared_indexes) > 0:
    result += float(1 / minimum_scared_ghost_distance)
return result + currentGameState.getScore()- 20 * len(capsules)
```

اولین تغییر اینست که دیگر نیازی نداریم با استفاده از ساکسسور فانکشن ها استیت بعد از اکشن را بدست اوریم و همه ی مواردی که لازم داریم را برای state فعلی به دست میاوریم.

دومین تغییر اینست که در بررسی غذا ها، اگر غذا به ما نزدیک باشد باید خورده شود. اینگونه پیاده سازیش میکنیم که هر چه غذا نزدیکتر باشد بدتر است و یا باید آنرا بخوریم یا دور شویم و میدانیم دور شدن نیز کار بهینه ای نیست پس به اندازه ی جمع مینیمم و ماکسیمم فاصله ها تا غذا ها از امتیاز کم میکنیم(یعنی برای بهبود امتیاز الان باید غذا را بخوری چون اگر ازش دور شوی باز هم امتیازت زیاد کم میشود) البته در ضریبی بزرگ نیز ضرب میکنیم تا تاثیر امتیاز خوردن را بیشتر کنیم(وقتی یک غذا را بخوریم مین و مکس ها جفتشون در حالت بعد زیاد میشوند و مقداری که کم میکنیم بیشتر میشود در حالی که صرفا 1 امتیاز بهمان اضافه شده).

تغییر دیگر اینست که کپسول ها را در بخش 1 در نظر نگرفتیم در صورتی که آنها 20 امتیاز به ما اضافه میکنند. به ازای وجود هر کپسول 20 امتیاز کم میکنیم این یعنی خوردن کپسول ها 40 امتیاز اضافه میکند و به نفعمان است(خودش 20 امتیاز دارد و یکی از 20 هایی که کم میکنیم هم کم میکند).

با این تغییرات که شرایط غذا ها و کپسول ها را بهتر در نظر گرفته ما به میانگین امتیاز بهتر و سرعت بیشتری دست پیدا میکنیم.

این تغییر ها ما را به نتایج زیر میرساند.

```
Average Score: 1214.0
Scores:        1175.0, 1365.0, 1160.0, 975.0, 1306.0, 1360.0, 1101.0, 1168.0, 1357.0, 1173.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
```

که همانطور که مشخص است در همه ی بازی ها برنده شده و میانگین هم از 1000 بیشتر است.