به نام خدا

پروژه hands on هوش مصنوعی بخش minimax

سارا رضائی منش ۸۱۰۱۹۸۵۷۶

هدف پروژه: استفاده از الگوریتم minimax برای شبیه سازی بازی چکرز

توضیح کلی پروژه: در این پروژه بازی چکرز را با دو agent هوشمند شبیه سازی می کنیم. فضا به صورت adversarial است و agent سفید به دنبال بیشینه کردن امتیاز خود و agent قرمز به دنبال کمینه کردن امتیاز حریف می باشد. همچنین مقدار زمان و حافظه مصرفی و همچنین میزان بهینگی تصمیم ها در عمق های مختلف بررسی و مقایسه می کنیم.

تابع getAllMoves و getValidMoves:

تابع getValidMoves تنها یک خط می شد و به همین علت برای کم کردن سربار صدا زدن تابع این تابع را حذف کردم و عمل گرفتن همه حرکات مجاز را در همان تابع minimax انجام دادم.

تابع getAllMoves به صورت زیر می باشد:

```
def getValidMoves(self, piece):
    pcolor, prow = piece.color, piece.row
    row, step = [max(prow-3, 0), -1] if (pcolor == RED) else [min(prow+3, ROWS), 1]

    left = self._traverseLeft(prow+step, row, step, pcolor, piece.col-1, skipped=[])
    right = self._traverseRight(prow+step, row, step, pcolor, piece.col+1, skipped=[])

    if piece.king:
        row, step = [max(prow-3, 0), -1] if (pcolor != RED) else [min(prow+3, ROWS), 1]
        left = {**left, **self._traverseLeft(prow+step, row, step, pcolor, piece.col-1, skipped=[])}
        right = {**right, **self._traverseRight(prow+step, row, step, pcolor, piece.col+1, skipped=[])}

    return {**right, **left}
```

در این تابع در صورتی که مهره قرمز باشد باید به سمت بالا صفحه و در غیر اینصورت باید به سمت پایین صفحه حرکت کنیم که این اصل در خط دوم تابع مشخص شده است. توابع _traverseLeft و _traverseRight را استفاده می کنیم تا حرکات مجاز سمت چپ و راست را بگیریم. در صورتی که مهره king باشد یعنی می توانیم هم حرکات رو به جلو و هم حرکات رو به عقب انجام دهیم. پس در صورت king بودن مهره، برای مقدار دیگر step و row نیز توابع پیمایش به راست و چپ را انجام می دهیم و مقدار بازگشتی آنها را به حرکات مجاز چپ و راست اضافه می کنیم. در پایان دیکشنری شامل همه حرکات مجاز چپ و راست برمی‌گردانیم.

تعریف evaluation:

در بسیاری از فضاها، branching factor و max-depth زیاد است و اگر بخواهیم که در هر مرحله درخت حالا را به صورت کامل تشکیل دهیم و به طور مطمئن حالات برد را تشخیص دهیم، باید زمان بسیار زیادی را منتظر پاسخ بمانیم که مطلوب نیست. به همین علت از تابع evaluation

استفاده می کنیم. این تابع به هر صفحه بر اساس معیارهایی که مهم هستند امتیازی نسبت می دهد. به عنوان مثال در تابع evaluation در بازی چکرز داریم:

```
def evaluate(self):
    return self.whiteLeft - self.redLeft + (self.whiteKings * 0.5 - self.redKings * 0.5)
```

یعنی در هر مرحله امتیاز یک رنگ به اختلاف تعداد مهره های آن رنگ با مهره های حریف است(مهره های king وزن متفاوتی دارند).

به این صورت می توانیم قبل از اینکه به حالت نهایی بازی برسیم می‌توانیم صفحه ها را با هم مقایسه کرده و صفحه با امتیاز بهتر را انتخاب کنیم.

شاید این انتخاب در کل بهترین انتخاب نباشد اما برای بدست آوردن پاسخ در زمان کوتاهتر می تواند از بخشی از این بهینگی چشم پوشی کرد.

تابع minimax:

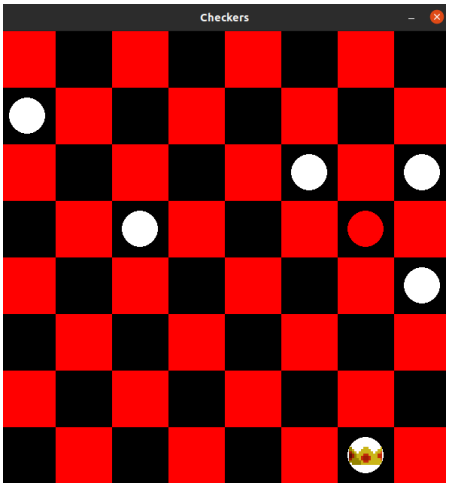
```
def minimax(position, depth, maxPlayer):
    if depth <= 0:
        return position.evaluate(), position

    pieceColor, v = RED, INF
    if maxPlayer:
        pieceColor, v = WHITE, NEG_INF

    ps = position.getAllPieces(pieceColor)
    best = deepcopy(position)
    for p in ps:
        moves = position.getValidMoves(p)

        for move in moves:
            bcopy = deepcopy(position)
            pcopy = bcopy.getPiece(p.row, p.col)
            bcopy = simulateMove(pcopy, move, bcopy, moves[move])
            nv, nb = minimax(bcopy , depth-1, not maxPlayer)
            if (maxPlayer and v < nv) or (not maxPlayer and v > nv):
                best, v = deepcopy(bcopy), nv
    return v, best


def simulateMove(piece, move, board, skips):
    board.remove(skips)
    board.move(piece, move[0], move[1])
    return board
```

تابع minimax یک تابع بازگشتی است که تا زمانی که به عمق تعیین شده برسد، درخت حالات را تشکیل می دهد و هنگام رسیدن به بیشترین عمق، تابع evaluation را روی صفحه به دست آمده انجام می دهد و آن صفحه و امتیاز آن را برمی‌گرداند. بدنه تابع به این صورت است که ابتدا رنگ مهره و مقدار اولیه v را مشخص می کند. در صورتی که رنگ مهره قرمز باشد یعنی باید به دنبال کمینه کردن امتیاز حریف باشیم پس v اولیه بی نهایت است و در صورتی که رنگ مهره سفید باشد یعنی باید به دنبال بیشینه کردن امتیاز خود باشیم و در نتیجه مقدار اولیه v باید منفی بی نهایت باشد. پس از مشخص کردن این مقادیر، همه مهره های همرنگ رنگی که نوبتش است را با استفاده از تابع getAllPieces انتخاب می کنیم. سپس به ازای هر مهره همه حرکاتی که می تواند انجام دهد را با استفاده از تابع getValidMoves می‌گیریم. به ازای هر حرکت مهره انتخاب شده، روی یک صفحه کپی شده(bcopy)، حرکت را با استفاده از تابع simulateMove پیاده سازی می کنیم و تابع minimax را برای صفحه جدید

بدست آمده و این دفعه با مهره حریف اجرا می کنیم. مقدار بازگشتی این تابع نشان دهنده بهترین انتخاب تا این مرحله است. در پایان با بررسی شرط آخر، در صورتی که مهره فعلی قرمز باشد(به دنبال کمینه کردن امتیاز حریف هستیم)، در صورتی که مقدار بازگشتی کمتر باشد، آن را با بهترین انتخاب قبل جایگزین می کنیم. برای سفید هم به همین صورت ولی با مقدار بیشتر عمل می‌کنیم. خروجی نهایی تابع بهترین انتخاب با توجه به عمق انتخاب شده است.

تست الگوریتم با عمق کم:

با توجه به تابع evaluation در عمق یک همیشه حالت زدن مهره حریف به نزدن آن و king شدن به حرکت عادی اولویت دارند. در تست مشاهده می شود که هر دو مهره این اولویت ها را قائل می شوند. سرعت محاسبات و حرکت مهره ها در این حالت بسیار بالاست و همچنین فضای کمی برای ساختن درخت حالات اشغال می شود اما مشخصا همیشه زدن مهره حریف بهترین حرکت نیست و ممکن است چند حرکت بعد به ضررمان تمام شود. پس بهتر است عمق تا جایی که زمان قابل قبول باشد، افزایش پیدا کند تا تصمیم گیری بهتری داشته باشیم. نتایج اجرا یا عمق یک برای هر دو مهره به صورت زیر می باشد:

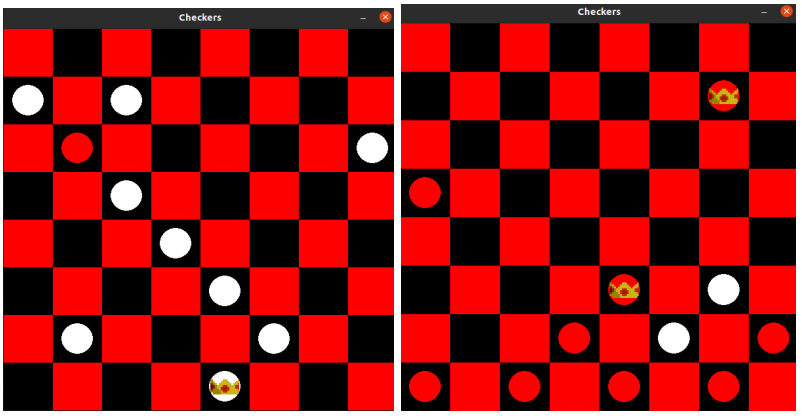


سفید به این دلیل نسبت قرمز عملکرد بهتری داشته که بررسی مهره ها هم برای سفید ها و هم برای قرمز ها از ردیف صفر شروع می شود. چون عمق کم است و دیدگاه خیلی آینده نگرانه نیست، اگر حالت زدن مهره حریف نداشته باشیم، سفید محتاطانه تر عمل می‌کند و مهره های عقبی را به جلو حرکت می دهد. در حالیکه قرمز اول مهره های جلویی اش بررسی می شود و به جلو حرکت داده می شوند. یعنی در بیشتر در معرض حذف شدن مهره هایش توسط حریف قرار می گیرد که در سناریویی که حریف در صورتی که بتواند همیشه می‌زند، این شیوه حرکت اصلا عاقلانه نیست. اگر بخواهیم بازی عادلانه تر پیش برود باید در صورت قرمز بودن مهره، بررسی را از آخرین ردیف جدول شروع کنیم.

تست الگوریتم با عمق بیشتر:

در این حالت رنگی که عمق بیشتری دارد، همه حرکات مهره دیگر را می‌تواند حدس بزند و حرکت بهتر از آن را انتخاب کند و به همین علت در تست بهتر عمل می‌کند و در نهایت برنده می شود و یا در صورت گیر کردن امتیاز بیشتری دارد(البته همیشه این سناریو برقرار نیست. چراکه مهره با

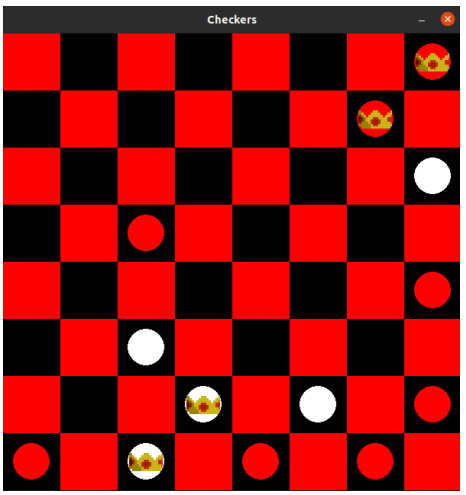
عمق پنج به نوعی میزان هوشمندی حریف خود را دست بالا می گیرد و بر اساس این انتظار پیش می رود در حالیکه حریف کم هوش تر است و شاید همیشه بهترین انتخاب تا عمق بررسی شده را انجام ندهد و این بررسی بیشتر در پایان بیشتر باعث ضرر شود) اما بررسی تا عمق پنج زمان و حافظه بسیار بیشتری نسبت به بررسی با عمق دو می گیرد. تصویر سمت راست، اجرای بازی با عمق ۵ برای مهره قرمز و عمق ۲ برای مهره سفید و تصویر سمت چپ اجرای بازی با عمق ۵ برای مهره سفید و عمق ۲ برای مهره قرمز می باشد.



تست الگوریتم با عمق برابر:

در این حالت هر دو مهره تصمیمات بهتری می گیرند اما زمان بسیار زیادی برای مشخص شدن نتایج طول می کشد. همچنین حافظه بسیار زیادی برای تشکیل دادن درخت حالات مصرف می شود.

نتیجه اجرا با عمق پنج برای هر دو رنگ به صورت نشان داده شده می باشد.



نتیجه گیری کلی:

در الگوریتم minimax، هر چه عمق بیشتر باشد، زمان و حافظه بیشتری برای انجام حرکت بعد صرف می شود اما تصمیم های بهتری گرفته می شود. پیش رفتن تا بیشترین عمق، زمان بسیار زیادی صرف می شود و به همین علت، از تابع evaluation برای مقایسه بورد ها در عمق های کوچکتر از بیشترین عمق استفاده می کنیم. اما حتی در صورت استفاده از این تابع نیز، از یک عمقی به بعد زمان و حافظه مصرفی ممکن است بیشتر از حالت مطلوب ما باشد. به همین علت باید با آزمون و خطا، تعادلی میان میزان دقت و زمان و حافظه مصرفی برقرار کنیم.

پیشنهادات:

ضمن تشکر از زحمات طراحان، به نظر بنده پروژه بسیار آموزنده ای بود و حالت گرافیکی آن هم خیلی جالب بود اما اگر ورودی های توابع traverse، خروجی آنها و کلاس ها بیشتر توضیح داده می‌شد، زمان بسیار کمتری صرف فهمیدن کد این بخش ها که کمک خاصی به فهم این مبحث نمی‌کنند می‌شد. من به شخصه برای اینکه بتوانم به درستی از این توابع به درستی استفاده کنم مجبور شدم منطق و خروجی آنها را تا حد خوبی مورد بررسی قرار دهم. همچنین داشتن یک اجرای درست هم می توانست در فرآیند دیباگ کمک کننده باشد.

منابع:

اسلاید های درس

جلسه های توجیهی