دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

درس هوش مصنوعی و کارگاه

گزارش ۲: پیاده‌سازی بازی Monopoly با الگوریتم جستجوی تخاصمی(لینک پروژه)

نگارش

کیارش مختاری دیزجی

۹۸۳۰۰۳۲

استاد اول

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

فروردین ۱۴۰۲

فهرست مطالب

صفحه

فهرست اشکال

صفحه

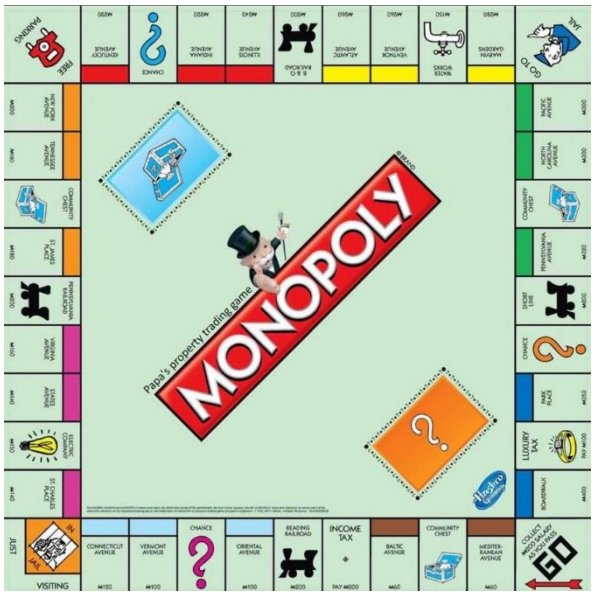
۱. فصل اول

مقدمه

مونوپولی یک بازی تخته ای اســـت که در این بازی هر شـــرکت کننده به کمک دو تاس در خانه‌های مختلف حرکت می‌کند و املاکی که در زمین بازی وجود دارد را خریداری می‌کند و تجارت خود را با خرید هتل گسترش می‌دهد. بازیکنان از حریف خود اجاره دریافت می‌کنند تا آنها را به سمت ورشکستگی بکشانند. برای هزینه پول‌ها نیز راه‌های مختلفی در این بازی وجود دارد.

۱- لوگوی بازی مونوپولی

هدف اصلی شرکت کنندگان در این بازی آن است که به ثروتمند ترین بازیکن تبدیل شوند و سایرین را به سمت ورشکستگی سوق دهند. البته در برخی از مواقع، برای بازی زمان تعیین می‌شود و در پایان آن زمان، برنده، ثروتمندترین فرد در بازی خواهد بود. به طور کلی بازی مونوپولی را می توان با ۲ تا ۶ نفر انجام داد که در این پیاده سازی صرفا از دو بازیکن استفاده شده است و همچنین تعداد ماکسیمم ۳۰ زاند بازی در نظر گرفته شده است.



۲- نمونه‌ای از برد بازی

۲. فصل دوم

قوانین حذف شده بازی

برای ساده سازی بازی برخی قوانین حذف شده‌اند:

۱. کارت ها حذف شده‌اند.

۲. هیچ خانه‌ای رنگ ندارد.

۳. مزایده و معاوضه نداریم.

۴. قانون سه بار تاس جفت پشت سرهم حذف شده است.

۵. در هر نوبت تنها یک کار انجام می‌دهیم.

۶. گزینه فروش املاک حذف شده است.

۷. و اگر بازکنی به خانه jail بیافتد تنها با دادن ۲۰۰ دلار می‌تواند خارج شود.

۳. فصل سوم

توضیح الگوریتم Expectiminimax

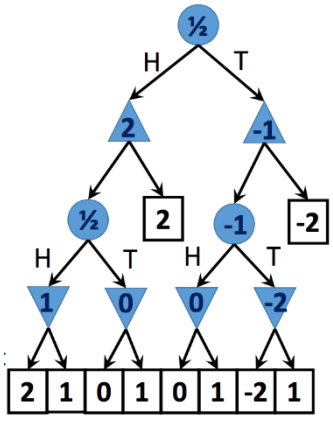
الگوریتم expectiminimax یک الگوریتم جستجو برای بازی های تصمیم گیری مشترک است که در آن باید با استفاده از روش حریصانه بهترین تصمیم را اتخاذ کنیم. این الگوریتم به دو بخش تقسیم می‌شود expecti وminimax :

در الگوریتمminimax ، یک بازیکن بازی را بر اساس امتیازی که به آن می‌دهد، ارزیابی می‌کند. سپس بازیکن دیگر نیز به همین کار می‌پردازد و این فرایند به صورت متناوب تکرار می‌شود تا تمامی حالت‌های بازی بررسی شوند و بهترین تصمیم برای بازیکن اول یا دوم به دست آید.

اما در بعضی از بازی‌ها، محیط بازی نیز به عنوان یک بازیکن می‌تواند عمل کند. به همین دلیل، الگوریتم expectiminimax برای بررسی این حالت‌ها استفاده می‌شود. در این الگوریتم، به جای مقدار دهی امتیاز به حالت‌های بازی، احتمالات برای هر کدام از اقدامات محیط در نظر گرفته می‌شود. سپس با استفاده از این احتمالات، مقدار انتظار برای امتیاز بازیکن محاسبه می‌شود. به این ترتیب، در هر مرحله، الگوریتم بهترین تصمیم را با توجه به احتمالات برای هر کدام از اقدامات در نظر می‌گیرد.

در کل، الگوریتم expectiminimax مشابه الگوریتم minimax است، با این تفاوت که به جای امتیاز هر حالت، مقدار انتظار امتیاز برای بازیکن محاسبه می‌شود. این الگوریتم برای حل بازی‌هایی با عوامل تصادفی، مانند بازی شطرنج با نرخ خطا، بازی های فضایی و ... بسیار مفید است.

در بازی مونوپولی نیز می‌توان از الگوریتم expectiminimax استفاده کرد. در این بازی، احتمال برای هر حرکت و موقعیت مالی متفاوت است و باعث شکل گیری عوامل تصادفی در بازی می‌شود .به عنوان مثال، در بازی مونوپولی، با استفاده از الگوریتم expectiminimax ، بازیکن می‌تواند تصمیم بگیرد که آیا باید خرید یک خانه جدید، ساخت یک خانه بر روی خانه‌هایی که در دست دارد، یا انتظار داشتن برای بلعیدن برای حریفان خود را انتخاب کند. با استفاده از این الگوریتم، بازیکن می‌تواند بهترین تصمیم را با توجه به احتمال برای هر گام در نظر بگیرد. به این ترتیب، بازیکن می‌تولند برای بردن بازی مونوپولی از بهترین تصمیم‌های ممکن بهره ببرد.



۳- نمونه مثالی از درخت الگوریتم **expectedminimax**

```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
            α := α + (Probability[child] × expectiminimax(child,
depth-1))
    return α
```

این کد یک تابع است که یک گره و عمق جاری را به عنوان ورودی دریافت می‌کند. اگر گره مسدود یا عمق مشخص شده برابر با صفر باشد، مقدار heuristic مربوط به گره را برمی‌گرداند. سپس، اگر رقیب در این گره بازی کند، حداقل مقدار را از فرزندان گره برمی‌گردانند. اگر ما در این گره بازی کنیم، حداکثر مقدار را از فرزندان گره برمی‌گرداند. اما اگر اتفاقی تصادفی در گره رخ دهد، میانگین وزن‌دار تمام فرزندان گره را برمی‌گرداند، که وزن هر یک از فرزندان با احتمال رخ دادن آن اتفاق تصادفی مربوطه تعیین می‌شود. در نهایت، تابع مقدار α را برمی‌گرداند که حاوی بهترین مقدار گره برای بازیکن فعلی است. این الگوریتم به طور کلی به صورت بازگشتی استفاده می‌شود و بازیکنان در هر مرحله از بازی از آن استفاده می‌کنند تا بهترین تصمیم را برای خود بگیرند.

٤. فصل چهارم

پیاده سازی بازی با استفاده از زبان پایتون

٤-١- فورمله کردن مسئله به شکل یک مسئله جستجو

برای پیاده سازی بازی تلاش شده است تا ابتدا مسئله به صورت زیر فورمله شود و سپس توابع کمکی که در کنار پیاده سازی توابع اصلی پیاده سازی شده‌اند. لازم به ذکر است که دیتای بازی در غالب یک دیکشنری در داخل کد ذخیره شده است.

Initial state, s_0:

Players, Players(s) denote whose turn is:

در کد به صورت یک کلاس stats تعریف شده است و در داخل دیکشنری players دو آبجکت از آن ذخیره شده است و مقدار دهی اولیه شده است.

Actions, Actions(s) available actions for the player:

```
def get_valid_actions(player):
     actions = []
     if properties[player.location]["name"] in ["Go", "Just Visiting/Jail",
"Free Parking"]:
          actions.append(all_actions[0])

     elif properties[player.location]["name"] == "Go To Jail":
          actions.append(all_actions[6])
     # Check if player is on a property
     elif properties[player.location]["type"] in ["property", "railroad",
"utility"]:
          # Check if the property is unowned
          if properties[player.location]["owner"] == "none":
               # Add the "BUY_PROP" action
               actions.append(all_actions[0]) # need to fix this(but latter)
               actions.append(all_actions[1])

          elif properties[player.location]["owner"] != player.id:
               # Add the "PAY_RENT" action
               actions.append(all_actions[2])
          elif properties[player.location]["type"] == "property":
               # Add the "BUILD_HOUSE" or "BUILD_HOTEL" action
               if player.balance >= properties[player.location]["hPrice"] and
properties[player.location]["houses"] < 4:
                    # Add the "BUILD_HOUSE" action
                    actions.append(all_actions[0])
                    actions.append(all_actions[4])

               # Check if player has enough money to build a hotel
               elif player.balance >= properties[player.location]["hPrice"]:
                    # Add the "BUILD_HOTEL" action
                    actions.append(all_actions[0])
                    actions.append( all_actions[5])

     # Check if player is on tax
     elif properties[player.location]["type"] == "tax":
          # Add the "PAY_TAX" action
          actions.append(all_actions[3])

     return actions
```

این تابع با اســـتفاده از موقعیت بازیکن اکشـــن‌هایی که یک بازیکن می‌تولند در هر حللت انجام دهد را به صورت یک لیست خروجی می‌دهد.

Transition model Result(s,a):

```
def transition(players: dict, properties: dict, current_player: int, action):
     new_properties = copy.deepcopy(properties)
     new_players = copy.deepcopy(players)

     # Execute action
     if action == all_actions[0]: # Do nothing
          pass

     elif action == all_actions[1]: # Buy property
          new_players[current_player].balance -=
new_properties[new_players[current_player].location]["price"]
          new_players[current_player].ownedP.append(new_players[current_playe
r].location)
          new_properties[new_players[current_player].location]["owner"] =
new_players[current_player].id

     elif action == all_actions[2]: # Pay rent
          charge_rent(new_players[current_player])

     elif action == all_actions[3]: # Pay tax
          new_players[current_player].balance -=
new_properties[new_players[current_player].location]["tax_price"]

     elif action == all_actions[4]: # Build house
          build_house(new_players[current_player],
new_players[current_player].location)

     elif action == all_actions[5]: # Build hotel
          build_hotel(new_players[current_player],
new_players[current_player].location)

     elif action == all_actions[6]: # Jail free
          new_players[current_player].location = 8
          new_players[current_player].balance -= 100

     return new_players, new_properties
```

این تابع متناسب با اکشن و بازیکن دریافتی آن را بروی بازیکن اعمال می‌کند و در آخر یک کپی از دیکشنری‌های players و properties را ریترن می‌کند.

Terminal test, Terminal−test(s):

```
def is_terminal(player):
     if player.balance <= 0:
         return True
     return False
```

این تابع نشان دهنده پایان بازی است که با مقایسه مقدار پول در دست بازیکن یک مقدار bool را ریترن می‌کند.

* لازم به ذکر است که توابع کمکی به صورت کامنت در داخل کد توضیح داده شده‌اند و همچنین نکات تکمیلی توابع اصلی نیر در داخل کد کامنت گذاشته شده اند.

تابع ارزشگذاری:

```
def evaluate_utility(player):
     net_worth = 0
     for prop_loc in player.ownedP:
          net_worth += properties[prop_loc]["price"]
          if properties[prop_loc]["hotels"] == 0:
               net_worth += properties[prop_loc]["rent"]+
10*properties[prop_loc]["houses"]
          else:
               net_worth += properties[prop_loc]["rent"]+
10*(properties[prop_loc]["houses"] + 1)

     for rr_loc in player.ownedRR:
          net_worth += properties[rr_loc]["price"]
          net_worth += properties[rr_loc]["rent"]

     for ut_loc in player.ownedUT:
          net_worth += properties[ut_loc]["price"]
          net_worth += properties[ut_loc]["rent"]

     net_worth += player.balance

     return net_worth
```

نحوه محاسبه تابع هیوریستیک از سرمایه بازیکن به علاوه‌ی مقدار پول نقد در دستش و اجاره حاصل از املاکی که دارد محاسبه می‌شود.

۵. فصل پنجم

نمونه اجرا بازی و بررسی منطقی بازی کردن آن

```
Round 0:
Player 1's turn
Player's status:
Player location: 0 , Player Balance: 1500 , Properties: [] , RailRoad: [] , Utility: []
Player 1 rolled 1 + 4 = 5
Player 1 chose action: BUY_PROP
================================================================================
Round 1:
Player 2's turn
Player's status:
Player location: 0 , Player Balance: 1500 , Properties: [] , RailRoad: [] , Utility: []
Player 2 rolled 1 + 2 = 3
Player 2 chose action: PAY_TAX
================================================================================
Round 2:
Player 1's turn
Player's status:
Player location: 5 , Player Balance: 1400 , Properties: [5] , RailRoad: [] , Utility: []
Player 1 rolled 5 + 6 = 11
Player 1 chose action: BUY_PROP
================================================================================
Round 3:
Player 2's turn
Player's status:
Player location: 3 , Player Balance: 1300 , Properties: [] , RailRoad: [] , Utility: []
Player 2 rolled 5 + 3 = 8
Player 2 chose action: BUY_PROP
================================================================================
Round 4:
Player 1's turn
Player's status:
Player location: 16 , Player Balance: 1200 , Properties: [5, 16] , RailRoad: [] , Utility: []
Player 1 rolled 5 + 4 = 9
Player 1 chose action: BUY_PROP
================================================================================
Round 5:
Player 2's turn
Player's status:
Player location: 11 , Player Balance: 1160 , Properties: [11] , RailRoad: [] , Utility: []
Player 2 rolled 4 + 6 = 10
Player 2 chose action: BUY_PROP
================================================================================
```

```
Round 18:
Player 1's turn
Player's status:
Player location: 28 , Player Balance: 40 , Properties: [5, 16, 25, 9, 14, 22, 28] , RailRoad: [] , Utility: []
Player 1 rolled 1 + 3 = 4
Player 1 chose action: PAY_TAX
Game over!
================================================================================
Final game state:
Player Balance: -60 , Properties: [5, 16, 25, 9, 14, 22, 28] , RailRoad: [] , Utility: []
Player Balance: 520 , Properties: [11, 21, 29, 2, 12] , RailRoad: [] , Utility: []
Player 1's net worth: 1668
Player 2's net worth: 1569
Time spend to finish the game: 0.14029479026794434
```

همانظور که در اسکرین شات هایی که از بخش‌های یک بار ران بازی مشاهده می‌کنید می‌توان فهمید که agent ها به صورت کاملا منطقی هر بار تصمیم به انجام کاری کرده‌اند مثلا در ابتدای بازی agent اول به دلیل نوع پیاده سازی تابع ارزش گذاری که از سرمایه بازیکن ها به علاوه‌ی مقدار پول نقد در دستش محاسبه می‌شود، اقدام به خرید یک ملک کرده است که کاری کاملا منطقی برای ابتدای بازی با توجه به تابع ارزش گذاری می‌باشد. البته لازم به ذکر است که بدلیل اینکه تابع ارزش گذاری به شدت ابتدایی می‌باشد و همینطور بسیاری از قوانین حذف شده‌اند دیده می‌شود که هیچ بازیکنی اقدام به خرید railroad و utility نکرده‌اند و این امر بدلیل موارد گفته شده کاملا طبیعی می‌باشد.

زمان اجرا بازی نیز در زمان معقولی به پایان رسیده است و با توجه به کم بودن عمق درخت به سرعت به پایان رسیده است اما با اضافه کردن عمق قطعا زمان به صورت نمایی رشد خواهد کرد.

منابع

* لازم به ذکر است که این پروژه با هکاری و **همفکری سیاوش پورفلاح و سایا هاشمیان** پیاده سازی شده است.

https://matiloos.com/blog/%D9%85%D9%88%D9%86%D9%88%D9%BE%D9%88%D9%84%DB%8C-%D8%B1%D8%A7-%D8%AF%D8%B1%D8%B3%D8%AA-%D8%A8%D8%A7%D8%B2%DB%8C%DA%A9%D9%86%DB%8C%D9%85/#1623524567540-6a0fb73d-3a74

https://en.wikipedia.org/wiki/Expectiminimax

دیتای بازی این ریپو جمع آوری شده است:

https://github.com/James0199/Textopoly/blob/main/monopoly.py

لینک فایل پیاده سازی بازی خودمان:

https://github.com/Kiarashmo/AI-course/blob/main/Project%203/main.py