دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

درس هوش مصنوعی و کارگاه

گزارش ۳: پیاده‌سازی بازی Pac-Man با استفاده از الگوریتم Minimax

نگارش

مهسا گودرزی

۹۹۱۲۰۴۳

استاد اول

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

آبان ۱۴۰۲

چکیده

یکی از انواع بازی‌ها، بازی‌های مجموع صفر هستند که این بازی‌ها اکیدا رقابتی هستند و عامل‌ها دارای سودمندی متضاد هستند. الگوریتم Minimax به عنوان روشی برای بهینه بازی کردن یک عامل معرفی می‌شود و به عامل کمک می‌کند به گونه‌ای بازی کند که بیشترین سودمندی را در برابر رقیبانش داشته باشد. در این مقاله به تحلیل کد نسخه‌ی ساده‌تر بازی Pac-Man پرداخته شده که در این کد الگوریتم Minimax با زبان پایتون پیاده‌سازی شده است و همچنین بررسی می‌شود که سودمندی پکمن به چه چیزهایی بستگی دارد و چه عمقی برای درخت بازی مناسب‌تر است.

واژه‌های کلیدی:

الگوریتم Minimax، بیشترین سودمندی، بازی Pac-Man، درخت بازی

فهرست مطالب

صفحه

فهرست اشکال

صفحه

فهرست جداول

صفحه

فصل اول

مقدمه

مقدمه

تئوری بازی‌ها[۱] شاخه‌ای از علم ریاضیات یا علم اقتصاد می‌باشد که به اثرات متقابل میان چند عامل[۲] می‌پردازد و روش‌هایی برای تشخیص رفتار بهینه تعیین می‌کند. تئوری بازی‌ها یکی از قدیمی‌ترین مباحثی است که در هوش مصنوعی راجع به آن زیاد مطالعه شده است؛ زیرا افراد زیادی بازی‌ها را دوست دارند و می‌توانند با آن‌ها کار کنند، بیان آن‌ها ساده است و عامل‌ها دارای تعداد اندکی عملیات هستند و معمولا بازی‌ها دارای یک توضیح واضح از محیط هستند.

ما می‌توانیم بعضی از بازی‌ها را به صورت مسئله‌ی جستجو شبیه سازی کنیم و در این حالت وضعیت صفحه‌ی بازی، وضعیت جستجوی مسئله است، حرکت‌های مجاز عمل کننده‌ها[۳] هستند و وضعیت‌های پایانی[۴]، وضعیت‌هایی هستند که در آن‌ها بازی را برده‌ایم یا بازنده شده‌ایم یا بازی بی‌نتیجه مانده است. می‌توانیم به بردن عدد ۱، به باختن عدد ۱-، و به بازی بی‌نتیجه مانده، عدد ۰ را نسبت دهیم.

در درخت بازی، ریشه‌ی درخت وضعیت اولیه است، سطح بعدی تمام حرکت‌های Max است، سطح بعدی تمام حرکت‌های Min است و این روند تکرار می‌شود. با حرکت‌های متناوب Min و Max سرانجام به وضعیت‌های پایانی می‌رسیم که می‌تواند با توجه به قوانین بازی به آن‌ها سودمندی نسبت داده شود. در روش Minimax از پردازنده‌ی جستجو برای پیدا کردن مسیر راه حل استفاده نمی‌کنیم، بلکه برای به دست آوردن دقیق‌ترین ارزیابی حرکت‌های ممکن استفاده می‌کنیم. Min و Max دو بازیکن هستند که Max می‌خواهد بازی را ببرد و Min هم می‌خواهد بازی را ببرد و برای این کار باید سودمندی Max را کاهش دهد و در واقع Max و Min رقیب هم هستند و هر دو بهترین بازی ممکن را انجام می‌دهند. در شکل ۱-۱ الگوریتم Minimax به صورت کلی توضیح داده شده است. در این الگوریتم، مقادیر Minimax به صورت بازگشتی محاسبه می‌شوند و مقادیر حالت‌های پایانی به عنوان بخشی از بازی داده شده‌اند.

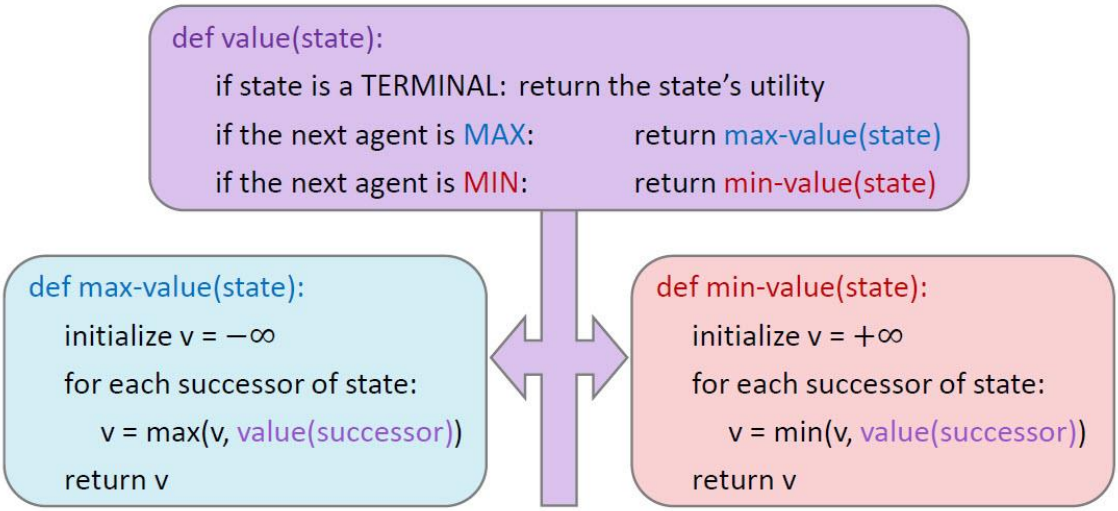
---

[۱] Game theory

[۲] Agent

[۳] Actuators

[۴] Terminal state



شکل ۱-۱- الگوریتم Minimax

مطابق با شکل بالا، به تابع value به عنوان ورودی وضعیت[۵] فعلی محیط بازی داده می‌شود، حال در خود تابع سه شرط اصلی وجود دارد؛ اگر این وضعیت، همان وضعیت پایانی تعریف شده در بازی باشد، تابع utility که یک تابع ارزش گذاری برای هر وضعیت بازی است، به عنوان خروجی برگردانده می‌شود. اگر در سطح بعدی درخت بازی، نوبت بازیکن Max باشد، تابع max-value برگردانده می‌شود و اگر در سطح بعدی درخت بازی، نوبت بازیکن Min باشد، تابع min-value به عنوان خروجی برگردانده می‌شود. در تابع max-value، ابتدا یک متغیر به نام v با مقدار ∞− تعریف می‌شود که منظور از v در اینجا، بیشترین سودمندی‌ای است که بازیکن Max می‌تواند داشته باشد. بعد از تعیین گره‌های فرزند[۶] Max، با کمک یک حلقه، هر یک از این گره‌های فرزند که در واقع هر کدام وضعیتی از محیط بازی هستند به عنوان ورودی به تابع value داده می‌شوند و خروجی آن با مقدار v مقایسه می‌شود و هر کدام که بزرگ‌تر باشد در v ذخیره می‌شود. سپس بعد از پایان حلقه مقدار v برگردانده می‌شود. در تابع min-value نیز روند کار مشابه است با این تفاوت که این تابع به عنوان خروجی کمترین سودمندی برای بازیکن Max را برمی‌گرداند و متغیر v در ابتدا نیز ∞+ تعریف می‌شود.

---

[۵] State

[۶] Successor

همانطور که گفته شد، تابع utility تابعی است که برای هر یک وضعیت‌های پایانی بازی مقداری را تعیین می‌کند و وقتی درخت بازی ما با عمق نامحدود باشد، این درخت وضعیت‌های ممکن بعد از وضعیت فعلی بازی را تا آخرین سطح عمق موجود پیش بینی می‌کند و سپس برگ‌های درخت بازی با کمک تابع utility ارزش گذاری می‌شوند، و الگوریتم Minimax بیشترین سودمندی را برای بازیکن Max پیدا می‌کند. اما اینکه در هر مرحله از بازی، درخت عمق نامحدودی داشته باشد، هزینه‌ی بسیار سنگینی می‌تواند داشته باشد و حافظه و زمان زیادی را می‌طلبد. به همین جهت ما یک تابع تخمین از utility به نام e-utility تعریف می‌کنیم که در این تابع ما ویژگی‌ها و حالت‌هایی را تعریف می‌کنیم که به هر وضعیت غیر پایانی بازی یک ارزش را اختصاص می‌دهند. وقتی عمق درخت ما محدود باشد، الگوریتم Minimax با کمک مقادیر ارزش دهی شده توسط تابع e-utility تصمیم می‌گیرد که تا آن عمق، چه وضعیتی برای بازیکن Max سودمندی بیشتری به همراه دارد. واضح است که هر چه تابع e-utility دقیق‌تر تعریف شود، بازیکن Max هم بهینه‌تر بازی می‌کند.

در این مقاله قصد داریم تا به شرح کد بازی Pac-Man با استفاده از الگوریتم Minimax بپردازیم. آن چه که به وسیله‌ی این کد پیاده‌سازی شده است، نسخه‌ای ساده از بازی Pac-Man می‌باشد؛ در یک صفحه‌ی ۹ × ۱۸ که تعدادی از خانه‌ها به صورت مانع تعریف شده‌اند و در بقیه‌ی خانه‌ها غذا وجود دارد، عامل پکمن باید در کمترین تعداد حرکات ممکن تمام این غذاها را بخورد و در طول مسیر خود به هیچ کدام از دو روحی که به صورت تصادفی در صفحه حرکت می‌کنند برخورد نکند؛ چرا که در صورت برخورد پکمن با یک روح، پکمن می‌بازد و بازی تمام می‌شود. الگوریتم Minimax به عامل پکمن کمک می‌کند که تا جای ممکن بهینه بازی کند و در هر مرحله به سمتی حرکت کند که بیشترین سود را برای او دارد.

فصل دوم

کد بازی Pac-Man

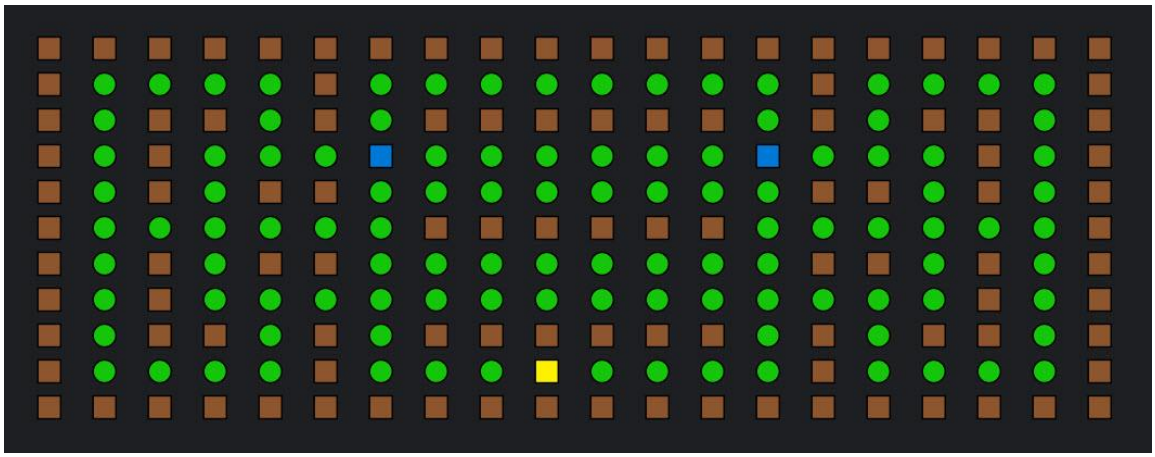
کد بازی Pac-Man

در این فصل کد بازی Pac-Man به صورت خلاصه توضیح و علت تعریف تابع‌های مختلف شرح داده می‌شود. کد این بازی در Pycharm که یک محیط توسعه یکپارچه[1] برنامه نویسی پایتون می‌باشد، نوشته شده است.

۲-۱- تعریف محیط بازی

همان طور که در مقدمه گفته شد، بازی مورد بحث در مقاله، یک نسخه‌ی ساده‌تر از نسخه‌ی اصلی بازی Pac-Man می‌باشد. صفحه‌ی اصلی بازی، یک صفحه‌ی ۹ × ۱۸ است، و شماره‌ی خانه‌ها به ترتیب از سمت چپ گوشه‌ی پایین از مختصات (۰, ۰) شروع می‌شود. ما به صورت دلخواه، مختصات شروع حرکت پکمن را (۸, ۰)، مختصات شروع حرکت روح اول را (۵, ۶) و مختصات شروع حرکت روح دوم را (۱۲, ۶) تعریف کرده‌ایم. همچنین تعدادی از خانه‌ها و همچنین کادر صفحه‌ی بازی را به عنوان مانع تعریف و در متغیر OBSTACLES ذخیره کرده‌ایم. مختصات غذاها نیز مختصات تمام خانه‌های صفحه به جز موانع و نقطه‌ی شروع حرکت پکمن می‌باشد که به صورت لیستی در متغیر food ذخیره شده‌اند. از آنجا که هر مرحله‌ی بازی در کنسول نمایش داده می‌شود، شکل ۲-۱ تصویری از محیط اولیه‌ی بازی نشان می‌دهد که در آن هیچ کدام از بازیکن‌ها هنوز شروع به حرکت نکرده‌اند و جزئیات بازی مطابق با توضیحات قبل است. در این تصویر پکمن با یک مربع زرد، روح‌ها با مربع آبی، غذاها با دایره‌ی سبز و موانع با مربع قهوه‌ای نشان داده شده‌اند. همچنین پکمن هر غذایی را که بخورد، در مرحله‌ی بعد در آن خانه به جای دایره سبز، دایره سفید نمایش داده می‌شود.

[1] Integrated Development Environment (IDE)



شکل ۲-۱- تصویری از محیط اولیه بازی

۲-۲- تابع e_utility و توابع مربوط به آن

همان طور که در فصل قبل اشاره شد، تابع e_utility یک تابع تخمین از utility است که به هر وضعیت غیر پایانی بازی یک امتیاز را نسبت می‌دهد. تعریف این تابع در عملکرد بازی نقش بسیار مهمی را ایفا می‌کند، زیرا الگوریتم Minimax بر حسب نوع امتیاز دهی این تابع است که تعیین می‌کند چه حرکتی برای پکمن بیشترین سود را دارد. الگوریتم Minimax تا عمقی که برای درخت بازی تعیین شده است پیش می‌رود و وقتی به آخر آن عمق می‌رسد، برای هر یک از برگ‌های درخت که هر کدام یک وضعیت از بازی را نشان می‌دهند، تابع e_utility را فراخوانی می‌کند تا به هر کدام از آن وضعیت‌ها یک امتیاز را اختصاص بدهد.

ورودی تابع e_utility، موقعیت پکمن، موقعیت روح اول، موقعیت روح دوم و لیست مختصات غذاهای باقی مانده است. در واقع این تابع به عنوان ورودی، جزئیات یک وضعیت از بازی را دریافت می‌کند. در اول این تابع، متغیری به نام score با مقدار ۰ تعریف شده است که مقدار این متغیر در ادامه‌ی عملکرد تابع بسته به حالت‌ها و شرط‌هایی که تعریف شده است، تغییر می‌کند و در نهایت این متغیر به عنوان خروجی برگردانده می‌شود. در شکل ۲-۲ کد این تابع نمایش داده شده است.

```python
def e_utility(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, food_position: list) -> int:
    score = 0

    if pacman_position == ghost1_position or pacman_position == ghost2_position:
        score -= 500

    if pacman_position in food_position:
        score += 20

    score += len(get_valid_moves(pacman_position)) * 5

    score += -20 * the_closest_food(pacman_position, food_position)

    score += 10 * the_closest_ghost(pacman_position, ghosts_position: [ghost1_position, ghost2_position])

    score += -15 * number_of_nearby_ghosts(pacman_position, ghosts_position: [ghost1_position, ghost2_position])

    score -= 1

    return score
```

شکل ۲-۲- کد تابع e_utility

همان‌طور که در کد این تابع دیده می‌شود، اولین حالتی که متغیر score دستخوش تغییر می‌شود وقتی است که موقعیت پکمن با موقعیت روح اول یا موقعیت روح دوم یکسان شود که بسیار حالت ناخوشایندی برای ما است و منجر به باخت پکمن می‌شود؛ در صورت وقوع چنین حالتی از score، ۵۰۰ امتیاز کسر می‌شود تا به دلیل این کسر امتیاز زیاد، الگوریتم Minimax وضعیتی را که منجر به این حالت می‌شود تا جای ممکن انتخاب نکند.

حالت بعدی در این تابع، این است که موقعیت پکمن با موقعیت یکی از غذاهای موجود در صفحه یکسان شود. به دلیل مطلوب بودن چنین حالتی، اگر این اتفاق بیفتد ۲۰ امتیاز به score اضافه می‌شود.

حالت دیگر در تابع e_utility این است که هر چه تعداد حرکات مجاز برای پکمن بیشتر باشد، مقدار score نیز افزایش می‌یابد. برای مثال در یک حالت پکمن ممکن است تنها مجاز به حرکت به سمت چپ یا راست یا صرفا ثابت ماندن در جای خود باشد، که این محدودتر از حالتی است که پکمن بتواند به چهار جهت بالا یا پایین یا چپ یا راست حرکت کند یا سر جای خودش باقی بماند و الگوریتم Minimax سعی می‌کند تا جای ممکن از انتخاب چنین حالت‌هایی که انتخاب زیادی برای حرکت پکمن وجود ندارد پرهیز کند. تعداد حرکات مجاز برای پکمن به کمک تابع get_valid_moves به دست می‌آید و به ازای هر حرکت مجاز، ۵ امتیاز به score اضافه می‌شود.

حالت بعدی در تابع e_utility این است که هر چقدر فاصله از غذا بیشتر باشد، امتیاز بیشتری نیز از score کم می‌شود، به همین جهت الگوریتم بازی مجبور به انتخاب وضعیتی است که پکمن را به غذای نزدیک‌تر برساند تا امتیاز کمتری از آن کم شود و سودمندی آن بیشتر شود. برای هر حالت ممکن در درخت بازی در تابع e_utility به ازای هر واحد فاصله با نزدیک‌ترین غذا، ۲۰ امتیاز از score کم شود. فاصله‌ی پکمن با نزدیک‌ترین غذا به وسیله‌ی تابع the_closest_food محاسبه می‌شود. در این تابع، که تصویر کد آن در شکل ۲-۳ آورده شده است، فاصله‌ی اقلیدسی پکمن با تمام غذاهای موجود در صفحه محاسبه و در نهایت کمترین آنها به عنوان خروجی برگردانده می‌شود که این مقدار کمترین فاصله‌ی پکمن با یک غذا است.

```
def the_closest_food(pacman_position: tuple, food_position: list) -> int:

    distance = min([sqrt(
        abs(pacman_position[0] - food_point_position[0]) ** 2 + abs(pacman_position[1] - food_point_position[1]) ** 2)
        for food_point_position in food_position])
    return int(distance)
```

شکل ۲-۳- کد تابع the_closest_food

حالت دیگر در تابع e_utility، توجه به فاصله‌ی پکمن تا نزدیک‌ترین روح است؛ بدین صورت که هر چقدر فاصله‌ی پکمن از یک روح بیشتر باشد، مقدار score نیز بیشتر می‌شود. در واقع به ازای هر واحد فاصله پکمن تا نزدیک‌ترین روح، ۱۰ امتیاز به score اضافه می‌شود، به همین جهت الگوریتم Minimax با توجه به این تعریف از تابع e_utility می‌کوشد که حرکتی را برای پکمن انتخاب کند که تا حد امکان از روح‌ها فاصله داشته باشد. محاسبه‌ی فاصله پکمن با نزدیک‌ترین روح به کمک تابع the_closest_ghost انجام می‌شود که تصویر کد آن در شکل ۲-۴ دیده می‌شود. این تابع موقعیت پکمن و موقعیت روح‌ها را به عنوان ورودی می‌گیرد و بعد از محاسبه‌ی فاصله اقلیدسی پکمن با هر کدام از روح‌ها، کمترین فاصله را به عنوان خروجی برمی‌گرداند.

```
def the_closest_ghost(pacman_position: tuple, ghosts_position: list) -> int:

    distance = min([sqrt(abs(pacman_position[0] - ghost[0]) ** 2 + abs(pacman_position[1] - ghost[1]) ** 2) for ghost in
                    ghosts_position])
    return int(distance)
```

شکل ۲-۴- کد تابع the_closest_ghost

تعداد روح‌ها در همسایگی پکمن، حالت دیگری است که در تابع e_utility لحاظ شده است. اگر در خانه‌های مجاور موقعیت پکمن، یک یا دو روح قرار گرفته باشند، برای مثال پکمن در بین هر دو روح گیر افتاده باشد، وضعیت بدی برای سودمندی پکمن به شمار می‌رود و احتمال باختن پکمن نیز بالا می‌رود. بنابراین باید تا جای ممکن از وقوع چنین حالتی جلوگیری کرد. پس به ازای هر روحی که در یک خانه‌ی مجاور موقعیت پکمن قرار بگیرد، ۱۵ امتیاز از score کم می‌شود. محاسبه‌ی تعداد روح‌ها در همسایگی پکمن به کمک تابع number_of_nearby_ghosts انجام می‌شود که تصویر کد آن در شکل ۲-۵ دیده می‌شود. این تابع موقعیت پکمن و روح‌ها را به عنوان ورودی می‌گیرد و با کمک حلقه، بررسی می‌کند که آیا هر کدام از روح‌ها در خانه‌های مجاور پکمن، که در واقع همان حرکات مجاز پکمن به شمار می‌روند، قرار می‌گیرد یا خیر. سپس تعداد روح‌هایی که در مجاورت پکمن قرار گرفته‌اند را برمی‌گرداند.

```python
def number_of_nearby_ghosts(pacman_position: tuple, ghosts_position: list) -> int:

    number_of_ghosts = 0
    for ghost in ghosts_position:
        if ghost in get_valid_moves(pacman_position):
            number_of_ghosts += 1
    return number_of_ghosts
```

شکل ۲-۵- کد تابع number_of_nearby_ghosts

آخرین حالتی که در تابع e_utility در نظر گرفته شده است، این است که به ازای هر حرکت پکمن، ۱ امتیاز از score کم شود. به همین جهت سعی می‌شود تا جای ممکن پکمن کمترین حرکات ممکن را داشته باشد.

در نهایت مقدار score به عنوان خروجی تابع e_utility برگردانده می‌شود. میزان وزن دهی به هر یک از حالت‌های این تابع، بر اساس اولویت‌بندی و همچنین آزمون و خطا بر اساس مقادیر مختلف بوده است.

۲-۳- الگوریتم Minimax و توابع مربوط به آن

همان‌طور که در مقدمه گفته شد، الگوریتم Minimax علاوه بر تابع e_utility از سه تابع اصلی دیگر نیز استفاده می‌کند؛ که در کد بازی Pac-Man توابع value و max_value و min_value می‌باشد.

۲-۳-۱- تابع value

تابع value موقعیت پکمن، موقعیت روح اول و روح دوم، عمق درخت بازی و نام بازیکن بعدی را به عنوان ورودی می‌گیرد. منظور از عمق یا همان depth در اینجا، عمقی از درخت بازی است که در آن به سر می‌بریم و با هر بار فراخوانی این تابع، یک سطح از عمق را پیش می‌بریم. همچنین برای تعیین این که در مرحله‌ی بعدی الگوریتم Minimax نوبت بازیکن Max است یا بازیکن Min، از ورودی next_agent استفاده می‌کنیم.

در این تابع که تصویر کد آن در شکل ۲-۶ آورده شده است، متغیر food که مختصات غذاهای باقی مانده است را به عنوان متغیری global شناسایی می‌کنیم. از آنجا که الگوریتم این تابع بازگشتی به شمار می‌رود، بنابراین سه شرط اصلی تعریف می‌کنیم.

```
def value(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, depth: int, next_agent: str) -> float:

    global food
    if depth == 0 or end_game(pacman_position, ghost1_position, ghost2_position, food) != 0:
        return e_utility(pacman_position, ghost1_position, ghost2_position, food)
    if next_agent == "PacMan":
        return max_value(pacman_position, ghost1_position, ghost2_position, depth)
    elif next_agent == "Ghost1" or next_agent == "Ghost2":
        return min_value(pacman_position, ghost1_position, ghost2_position, depth, current_agent=next_agent)
```

شکل ۲-۶- کد تابع value

شرط اول آن است که اگر به انتهای عمق درخت رسیده باشیم یا به وضعیتی از بازی رسیده باشیم که به عنوان وضعیت پایانی بازی به شمار می‌رود، تابع e_utility برگردانده می‌شود که کارکرد آن را در بخش قبلی توضیح دادیم. در واقع در این مرحله با کمک تابع e_utility به برگ‌های عمق انتخابی درخت امتیازی داده‌ایم، و سپس به دلیل بازگشتی بودن تابع، الگوریتم Minimax از این امتیازها برای محاسبه‌ی سودمندی عامل پکمن استفاده می‌کند.

شرط دوم آن است که اگر ورودی next_agent برابر با PacMan باشد، یعنی در مرحله‌ی بعدی در درخت بازی نوبت عامل پکمن باشد، تابع max_value برگردانده می‌شود که این تابع بیشترین ارزش گره‌های فرزندش را برمی‌گرداند.

شرط سوم نیز آن است که اگر ورودی next_agent برابر با Ghost۱ یا Ghost۲ باشد، یعنی در مرحله‌ی بعدی در درخت بازی نوبت عامل روح اول یا روح دوم باشد، تابع min_value برگردانده

می‌شـــود که این تابع کمترین ارزش گره‌ی فرزندانش را برمی‌گردلند و در واقع به دنبال این اســـت که ســـودمنـدی پکمن را کـاهش دهـد. همچنین ورودی next_agent در این تـابع بـه عنوان ورودی current_agent داده می‌شـود تا مشـخص شـود که دقیقا در این مرحله منظور ما روح اول اسـت یا روح دوم و در تابع min_value هر کدام از این دو، کارکرد مخصوص به خود را دارند.

۲-۳-۲- تابع max_value

تابع max_value در واقع مربوط به مرحله‌ی محاســـبه‌ی بیشـــترین ســـودمندی در گره‌های فرزند در الگوریتم Minimax اســـت. این تابع به عنوان ورودی، موقعیت پکمن، موقعیت روح اول و روح دوم، و همچنین عمقی از درخت بازی که در آن هستیم را دریافت می‌کند. با توجه به کد این تابع که تصویر آن در شـکل ۲-۷ آورده شـده اسـت، به دلیل آن که این تابع در اصـل بیشـترین ارزش گره‌های فرزند عامل پکمن را بـه عنوان خروجی برمی‌گردانـد، پس مـا تمـام گره‌هـای فرزنـد پکمن را بـا کمـک تـابع get_valid_moves در مـتـغـیـر successors ذخـیـره مـی‌کـنـیـم. هـمـچـنـیـن مـتـغـیـری بـا نـام maximum_value با مقدار اولیه‌ی ∞− تعریف می‌کنیم که با مقایســـه مقدار این متغیر با ارزش هر یک از گره‌های فرزند پکمن، در نهایت این متغیر به عنوان خروجی برگردانده می‌شود.

```python
def max_value(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, depth: int) -> float:

    successors = get_valid_moves(pacman_position)
    maximum_value = float('-inf')

    for successor in successors:
        new_pacman_position = successor
        score = value(new_pacman_position, ghost1_position, ghost2_position, depth=depth - 1, next_agent="Ghost1")
        maximum_value = max(maximum_value, score)

    return maximum_value
```

شکل ۲-۷- کد تابع max_value

با اجرای یک حلقه for بر روی successors که تمام وضـــعیت‌های ممکن برای حرکت بعدی پکمن هســـتند، متغیری به نام score تعریف می‌کنیم که در هر بار اجرای این حلقه، هر کدام از گره‌های فرزند پکمن را به عنوان ورودی جدید به تابع value می‌دهد و همچنین عمق درخت را یک ســـطح جلوتر می‌برد و عامل بعدی را روح اول به این تابع معرفی می‌کند. سـپس مقدار score هر چه که شـد، با مقدار

maximum_value مقایسه می‌شود و هر کدام که بزرگ‌تر باشد در این متغیر ذخیره می‌شود و پس از پایان حلقه maximum_value به عنوان خروجی برگردانده می‌شود.

۲-۳-۳- تابع min_value

تابع min_value در واقع مربوط به مرحله‌ی محاسبه‌ی کمترین سودمندی در گره‌های فرزند در الگوریتم Minimax است. این تابع به عنوان ورودی، موقعیت پکمن، موقعیت روح اول و روح دوم، عمقی از درخت بازی که در آن هستیم و همچنین نام عامل فعلی را دریافت می‌کند. با توجه به کد این تابع که تصویر آن در شکل ۲-۸ آورده شده است، ابتدا متغیری با نام minimum_value با مقدار اولیه‌ی $+\infty$ تعریف می‌کنیم که با مقایسه مقدار این متغیر با ارزش هر یک از گره‌های فرزند روح اول یا روح دوم، در نهایت این متغیر به عنوان خروجی برگردانده می‌شود.

```
def min_value(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, depth: int,
              current_agent: str) -> float:

    minimum_value = float('inf')

    if current_agent == "Ghost1":
        successors = get_valid_moves(ghost1_position)

        for successor in successors:
            new_ghost1_position = successor
            score = value(pacman_position, new_ghost1_position, ghost2_position, depth=depth - 1, next_agent="Ghost2")
            minimum_value = min(minimum_value, score)

    elif current_agent == "Ghost2":
        successors = get_valid_moves(ghost2_position)

        for successor in successors:
            new_ghost2_position = successor
            score = value(pacman_position, ghost1_position, new_ghost2_position, depth=depth - 1, next_agent="PacMan")
            minimum_value = min(minimum_value, score)

    return minimum_value
```

شکل ۲-۸- کد تابع min_value

از آنجا که در الگوریتم Minimax این بازی، ما دو بازیکن برای سطح‌های Min درخت بازی داریم، بنابراین باید مشخص کنیم که در این مرحله منظور ما کدام یک از دو عامل روح اول یا روح دوم می‌باشد که این کار به وسیله‌ی ورودی current_agent مشخص می‌شود. در واقع اگر هر کدام از این دو روح به عنوان عامل فعلی در این سطح از درخت شناسایی شوند، عملکرد تابع به صورت کلی برای هر دو یکسان

اســـت ولی با این تفاوت که حلقه‌ی for به طور مجزا برای هر کدام روی گره‌های فرزند آن عامل اعمال می‌شـــود و در تعیین عامل بعدی برای تابع value، اگر عامل فعلی روح اول باشـــد عامل بعدی روح دوم اسـت، و اگر عامل فعلی روح دوم باشـد عامل بعدی پکمن می‌باشـد. در کل عملکرد این تابع مشـابه تابع max_value اســـت ولی به جای ماکزیمم گرفتن از ارزش گره‌های فرزند عامل فعلی، مینیمم آن‌ها محاسبه و در متغیر minimum_value ذخیره و در نهایت به عنوان خروجی برگردانده می‌شود.

۲-۳-٤- تابع end_game

تابع end_game به هر وضـعیت پایانی بازی عددی را نسـبت می‌دهد. همان طور که تصـویر کد این تابع در شـکل ۲-۹ آورده شـده اسـت، این تابع به عنوان ورودی موقعیت پکمن، موقعیت روح اول و روح دوم و فهرســـت مختصـــات غذاهای باقی مانده را دریافت می‌کند. حال اگر موقعیت پکمن با موقعیت یکی از دو روح یکسان شود، به این معنی است که پکمن باخته است و عدد ۱- به عنوان خروجی برگردانده می‌شود. اگر تعداد غذاهای باقی مانده به صـفر برسـد، یعنی پکمن برده اسـت و عدد ۱ به عنوان خروجی برگردانده می‌شود. اگر هیچ کدام از این دو حالت اتفاق نیفتد، عدد ۰ برگردانده می‌شود که یعنی هنوز بازی به هیچ کدام از حالات پایانی خود نرسیده است.

```python
def end_game(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, current_food: list) -> int:

    if pacman_position == ghost1_position or pacman_position == ghost2_position:
        return -1
    if len(current_food) == 0:
        return 1
    return 0
```

شکل ۲-۹- کد تابع end_game

۲-٤- توابع دیگر بازی

علاوه بر توابعی که تا الان درباره‌ی آن‌ها صـحبت شـد، در کد بازی از چند تابع دیگر نیز اسـتفاده شـده است که وجود آن‌ها برای اجرای درست بازی ضروری است.

۲-٤-۱- تابع get_valid_moves

تابع get_valid_moves که پیش از این در الگوریتم Minimax از آن حرف زده شده بود، موقعیت یک عامل را به عنوان ورودی می‌گیرد و تعداد حرکات مجاز بعدی برای این عامل را به صورت یک لیست به عنوان خروجی برمی‌گرداند. در کد این تابع که تصویر آن در شکل ۲-۱۰ آورده شده است، حداکثر ۵ حرکت مجاز برای موقعیت بعدی یک عامل در نظر گرفته شده است؛ اگر در خانه‌ی بالایی عامل مانعی نباشد می‌تواند به سمت بالا حرکت کند، اگر در خانه‌ی پایینی عامل مانعی نباشد می‌تواند به سمت پایین حرکت کند، اگر در خانه‌ی راستی عامل مانعی نباشد می‌تواند به سمت راست حرکت کند، اگر در خانه‌ی چپی عامل مانعی نباشد می‌تواند به سمت چپ حرکت کند و یا می‌تواند بر سر جای خود ثابت باقی بماند. هر کدام از این حرکات که مجاز باشند به لیست valid_moves اضافه می‌شوند و در نهایت این لیست به عنوان خروجی برگردانده می‌شود.

```python
def get_valid_moves(agent_position: tuple) -> list:

    x, y = agent_position
    valid_moves = []

    # Move Up
    if (x, y + 1) not in OBSTACLES:
        valid_moves.append((x, y + 1))
    # Move Down
    if (x, y - 1) not in OBSTACLES:
        valid_moves.append((x, y - 1))
    # Move Right
    if (x + 1, y) not in OBSTACLES:
        valid_moves.append((x + 1, y))
    # Move Left
    if (x - 1, y) not in OBSTACLES:
        valid_moves.append((x - 1, y))

    # Agent Staying In Place
    valid_moves.append(agent_position)

    return valid_moves
```

شکل ۲-۱۰- کد تابع get_valid_moves

۲-٤-۲- تابع display_game_screen

این تابع وضعیت صفحه‌ی بازی را در هر مرحله نشان می‌دهد. با توجه به کد این تابع که تصویر آن در شکل ۲-۱۱ آورده شده است، این تابع موقعیت فعلی پکمن، موقعیت فعلی روح اول و روح دوم و همچنین مختصات غذاهای باقی مانده را به عنوان ورودی دریافت می‌کند، سپس با اجرای یک حلقه بر روی تمام خانه‌های صفحه، خانه‌هایی که مانع باشند با مربع قهوه‌ای، خانه‌هایی دو روح در آن قرار گرفته باشند با مربع آبی، خانه‌ای که پکمن در آن قرار گرفته باشد با مربع زرد، خانه‌هایی که در آن‌ها غذاهای خورده نشده باشد با دایره سبز و در نهایت خانه‌هایی که خالی و بدون غذا باشند با دایره سفید نمایش داده می‌شوند.

```python
def display_game_screen(pacman_position: tuple, ghost1_position: tuple, ghost2_position: tuple, food_position: list):

    for j in range(9, -2, -1):
        line = ""
        for i in range(-1, 19):
            if (i, j) in OBSTACLES:
                line += " 🟫 "
            elif (i, j) == ghost1_position or (i, j) == ghost2_position:
                line += " 🟦 "
            elif (i, j) == pacman_position:
                line += " 🟨 "
            elif (i, j) in food_position:
                line += " 🟢 "
            else:
                line += " ⚪ "
        print(line)
    print()
```

شکل ۲-۱۱- کد تابع display_game_screen

۲-٤-۳- تابع play_game

این تابع، تابع اصلی اجرای این بازی است. این تابع دو ورودی می‌گیرد؛ یکی depth و یکی call_sleep_fun. در depth ما عمق دلخواه درخت بازی را تعیین می‌کنیم و در مورد call_sleep_fun، اگر مقدار True را به آن بدهیم، تابع sleep با ورودی ۱ ثانیه از کتابخانه time فراخوانده می‌شود و در هر بار اجرای حلقه‌ی اصلی بازی، این حلقه با یک ثانیه تاخیر اجرا می‌شود. در واقع کاربرد این ورودی برای وقتی است که کاربر می‌خواهد با سرعت کمتر و دقت بیشتری تغییرات مربوط به صفحه‌ی بازی را مشاهده کند.

برای محاسبه‌ی زمان اجرای بازی، یک متغیر به نام start_time در ابتدا و یک متغیر به نام end_time در انتهای تابع play_game تعریف می‌کنیم که هر دو متغیر به کمک متد datetime.now() از کتابخانه datetime زمان شروع و پایان بازی را محاسبه می‌کنند و تفاضل این دو متغیر به عنوان زمان اجرای بازی در متغیر execution_time ذخیره می‌شود.

در ابتدای تابع play_game متغیر food به عنوان متغیری global شناخته می‌شود تا بتوان بر روی آن تغییرات لازم را اعمال کرد. سپس با چند دستور print، مشخصات اولیه بازی از جمله لوگوی بازی، موقعیت مانع‌ها، موقعیت غذاها، نقطه‌ی شروع حرکت پکمن و روح‌ها نمایش داده می‌شود. سپس متغیرهایی به نام‌های pacman_position، ghost۱_position، ghost۲_position تعریف می‌شوند که به ترتیب موقعیت‌های فعلی پکمن و روح اول و روح دوم هستند و به عنوان مقداردهی اولیه، موقعیت اولیه هر کدام از این عامل‌ها به آن‌ها داده می‌شود. سپس متغیری به نام current_score و level با مقداردهی اولیه ۰ تعریف می‌شوند که هر کدام به ترتیب امتیاز کسب شده تا الان و مرحله‌ی کنونی را مشخص می‌کنند.

سپس نوبت به حلقه‌ی اصلی بازی می‌رسد که یک حلقه‌ی while است و تا زمانی ادامه می‌یابد که خروجی تابع end_game بر اساس موقعیت فعلی عامل‌ها برابر با صفر باشد. یعنی تا زمانی که بازی به وضعیت پایانی خود نرسیده باشد، این حلقه اجرا می‌شود. در ابتدای این حلقه در هر بار اجرا، شماره مرحله‌ی بازی، امتیاز کسب شده تا الان، تعداد غذاهای باقی مانده، موقعیت پکمن و دو روح و همچنین صفحه‌ی بازی که به کمک تابع display_game_screen آن را نمایش می‌دهیم، بر روی کنسول چاپ می‌شوند.

بعد از این، در این حلقه عملیاتی که در تابع max_value داشتیم را بازنویسی می‌کنیم، زیرا می‌خواهیم پکمن به عنوان بازیکن Max ما شناخته شود و در واقع در ریشه‌ی درخت بازی قرار بگیرد. در واقع در هر بار اجرای این حلقه، یک درخت بازی جدید بسته به موقعیت‌های فعلی پکمن و روح‌ها ساخته می‌شود و پکمن به عنوان ریشه‌ی این درخت در الگوریتم Minimax معرفی می‌شود. پس در این مرحله ما به کمک تابع get_valid_moves، موقعیت‌های مجاز بعدی پکمن و در واقع گره‌های فرزند آن را در متغیری به نام successors ذخیره می‌کنیم، با اجرای حلقه‌ی for بر روی successors، الگوریتم Minimax را برای به دست آوردن سودمندترین حرکت پکمن اعمال می‌کنیم. همان‌طور که گفته شد،

این قسمت از حلقه دقیقا مانند عملکرد تابع max_value می‌باشد با این تفاوت که ما بعد از پیش بینی بهترین امتیاز ممکن برای پکمن در حرکت بعدی خود، بهترین حرکت مجاز را هم در متغیری به نام best_move ذخیره می‌کنیم و سپس به عنوان موقعیت جدید پکمن آن را در نظر می‌گیریم. کد این قسمت از بازی در شکل ۲-۱۲ آورده شده است.

```
depth = depth
successors = get_valid_moves(pacman_position)
best_move = None
maximum_value = float('-inf')

for successor in successors:
    new_pacman_position = successor
    score = value(new_pacman_position, ghost1_position, ghost2_position, depth - 1, next_agent="Ghost1")
    if score > maximum_value:
        maximum_value = score
        best_move = new_pacman_position

pacman_position = best_move
```

شکل ۲-۱۲- کد تعیین حرکت بعدی پکمن

بعد از این، نوبت به تغییر امتیاز پکمن در این مرحله است. به ازای هر حرکت پکمن، یک امتیاز از current_score کسر می‌شود و اگر پکمن هر یک از غذاها را بخورد، ۱۰ امتیاز به current_score اضافه می‌شود و مختصات آن غذا از food حذف می‌شود. تصویر این بخش از کد در شکل ۲-۱۳ آورده شده است.

```
current_score -= 1
if pacman_position in food:
    food.remove(pacman_position)
    current_score += 10
```

شکل ۲-۱۳- کد محاسبه امتیاز فعلی پکمن

سپس نوبت تعیین موقعیت‌های جدید دو روح می‌باشد. همان طور که در مسئله‌ی بازی مطرح شده بود، این دو روح حرکاتی تصادفی دارند، به همین خاطر به کمک تابع choice از کتابخانه random، یکی از حرکات مجاز آنها که به وسیله‌ی تابع get_valid_moves مشخص شده است، به صورت تصادفی

انتخاب و به عنوان موقعیت جدید روح‌ها در متغیر مربوط به آن‌ها ذخیره می‌شود. در شکل ۲-۱۴ تصویر کد این بخش از بازی آورده شده است.

```
ghost1_position = choice(get_valid_moves(ghost1_position))
ghost2_position = choice(get_valid_moves(ghost2_position))
```

شکل ۲-۱۴- کد تعیین حرکت بعدی روح‌ها

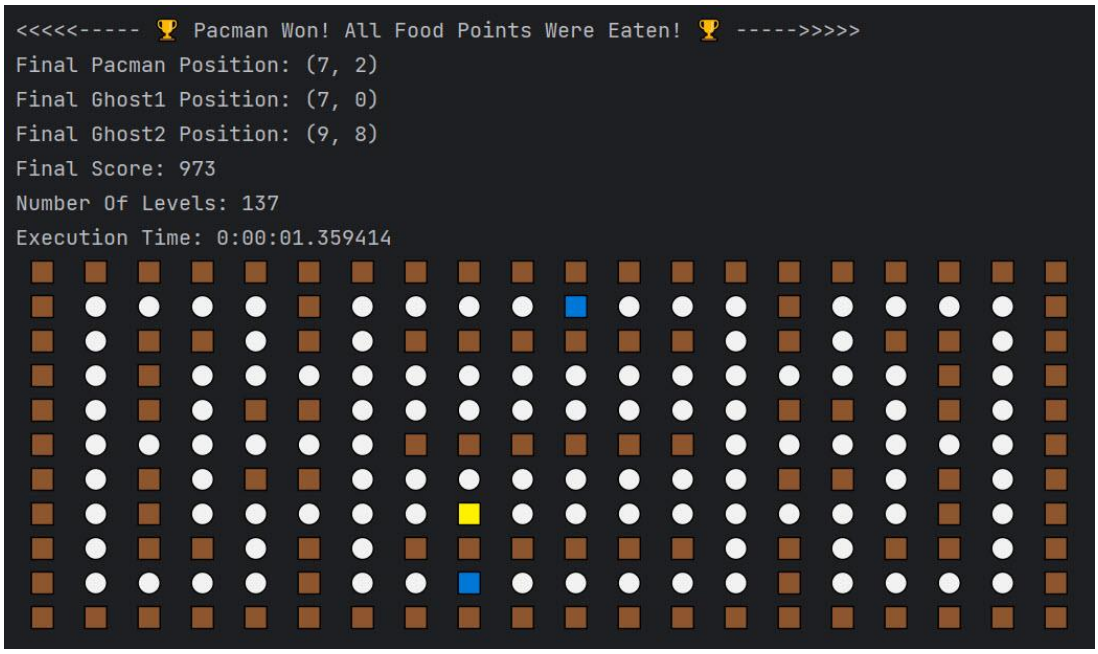
بعد از اجرای این بخش از کد، بررسی می‌شود که آیا بازی به وضعیت پایانی خود رسیده است یا خیر، بر حسب خروجی تابع end_game، اگر پکمن بازی را برده باشد یک پیام بر روی صفحه چاپ می‌شود و برد پکمن را اعلام می‌کند، در غیر این صورت اگر پکمن باخته باشد، یک پیام بر روی صفحه چاپ می‌شود و باخت پکمن را اعلام می‌کند. تصویر این بخش از کد در شکل ۲-۱۵ دیده می‌شود.

```
if end_game(pacman_position, ghost1_position, ghost2_position, food) == 1:
    print("<<<<<----- 🏆 Pacman Won! All Food Points Were Eaten! 🏆 ----->>>>>")
elif end_game(pacman_position, ghost1_position, ghost2_position, food) == -1:
    print("<<<<<----- 👾 Oops! Pacman Was Eaten By A Ghost! 👾 ----->>>>>")
```

شکل ۲-۱۵- اعلام برد یا باخت پکمن

بعد از این مقدار level که نشان دهنده‌ی مرحله‌ی بازی است یک عدد افزایش می‌یابد و سپس حلقه تمام می‌شود. حال بسته به مقدار خروجی تابع end_game این حلقه می‌تواند دوباره اجرا شود. بعد از پایان اجرای حلقه، که به معنای رسیدن بازی به یکی از حالات پایانی خود است، موقعیت نهایی پکمن و روح‌ها، امتیاز نهایی پکمن، تعداد مراحل اجرای بازی، زمان اجرای بازی و صفحه‌ی وضعیت پایانی بازی بر روی کنسول چاپ می‌شوند و سپس کار تابع play_game تمام می‌شود.

حال برای اجرای بازی تنها کافی است که تابع play_game را با ورودی‌های لازم فراخوانی کنیم. برای نمونه ما در یک بار اجرای این بازی، عمق را ۴ در نظر می‌گیریم و به call_sleep_fun هم مقدار False را می‌دهیم. نتیجه‌ی این بازی در شکل ۲-۱۶ دیده می‌شود. در این بازی، پکمن در ۱۳۷ مرحله و در مدت زمان حدود ۱.۳۶ ثانیه، تمام غذاها را خورده و بازی به نفع پکمن تمام شده است که نتیجه‌ی بسیار مطلوبی را به ما می‌دهد.



شکل ۲-۱۶- نمونه‌ای از یک وضعیت پایانی بازی

فصل سوم

تحلیل الگوریتم Minimax در بازی

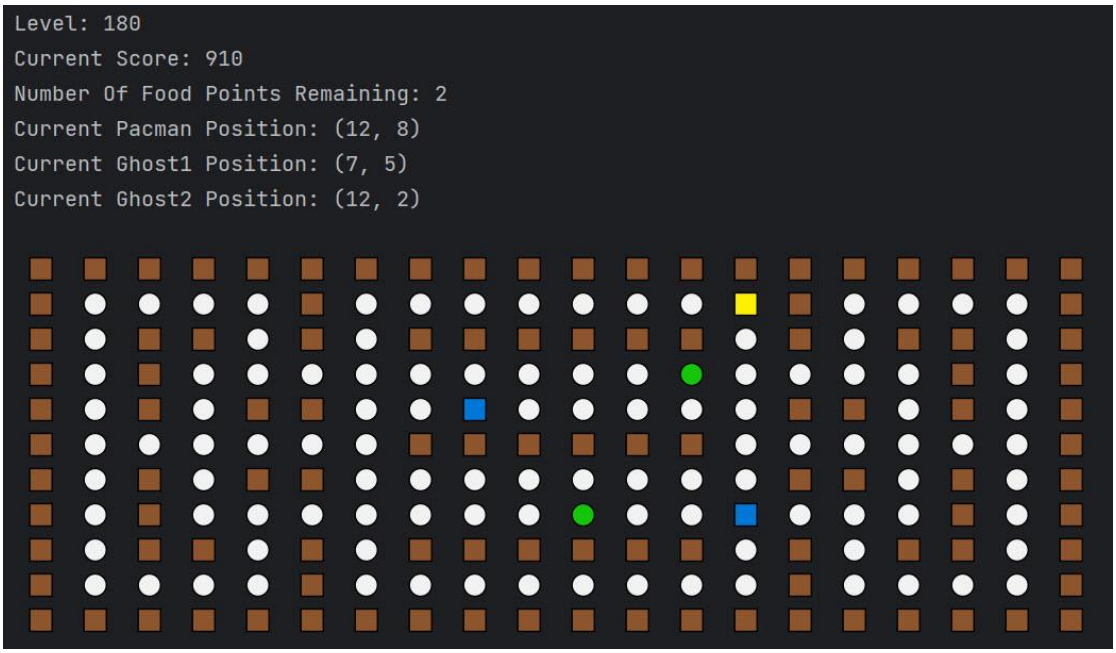
تحلیل الگوریتم Minimax در بازی

در این فصـــل به تحلیل و بررســـی بیشـــتر الگوریتم Minimax در بازی Pac-Man می‌پردازیم؛ نحوه انتخاب عامل بر اســـاس این الگوریتم را در یک وضـــعیت توضـــیح می‌دهیم، حرکات پکمن را با توجه به الگوریتم تحلیل می‌کنیم و علت انتخاب عمق درخت را شرح می‌دهیم.

۳-۱- نحوه انتخاب عامل بر اساس الگوریتم Minimax

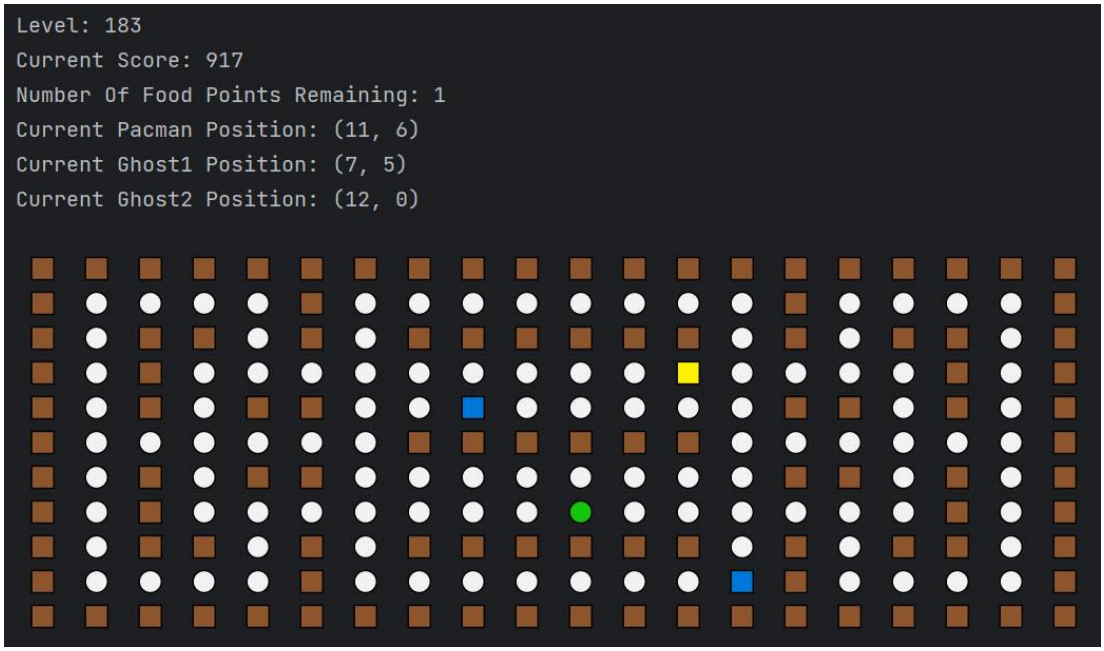
در فصل قبل به بررسی کد بازی پرداختیم و گفتیم که تابع e_utility بر اساس چه معیارها و حالت‌هایی تعریف می‌شـــود. آنچه که در انتخاب حرکات بعدی پکمن توســـط الگوریتم Minimax اهمیت دارد این اســـت که در کمترین تعداد حرکات ممکن تمام غذاها خورده شـــوند، پکمن از روح‌ها فاصـــله‌ی مناســـبی داشته باشد که در حرکات بعدی به آن‌ها برخورد نکند، و به دنبال نزدیک‌ترین غذاها برود.

به عنوان مثال، در یکی از اجراهای بازی، به وضـــعیتی مانند وضـــعیت شـــکل ۳-۱ برمی‌خوریم. در این وضعیت پکمن در خانه‌ی (۱۲, ۸) قرار دارد و روح اول و دوم به ترتیب در خانه‌های (۷, ۵) و (۱۲, ۲) قرار دارند و تنها دو غذای دیگر در نقاط (۱۱, ۶) و (۹, ۲) قرار دارند.



شکل ۳-۱- پکمن در خانه‌ی (۱۲, ۸)

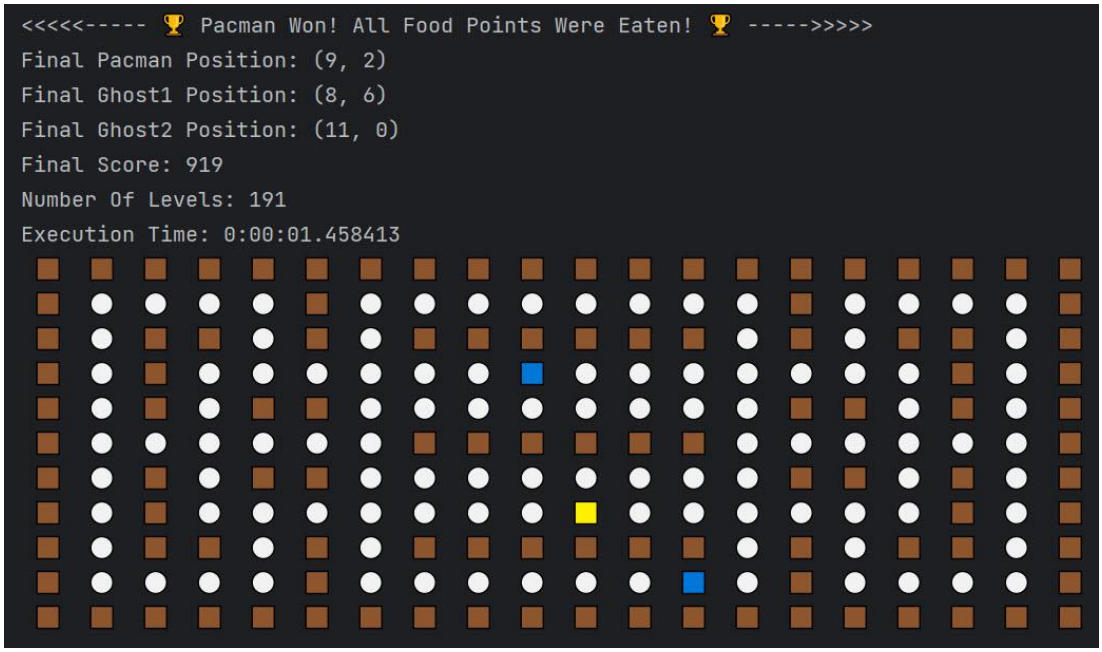
الگوریتم Minimax با توجه به نوع تعریف تابع e_utility حرکتی را برای پکمن انتخاب می‌کند که او را به نزدیک‌ترین غذا، یعنی (۱۱, ۶) برســاند. همچنین در این حرکات فاصــله‌ی پکمن با روح‌ها نیز در نظر گرفته می‌شود و برای پکمن سودمندتر است که در کمترین حرکات ممکن به نقطه (۱۱, ۶) برسد. وقتی پکمن در موقعیت (۱۲, ۸) اسـت، سـه انتخاب برای حرکت بعدی خود دارد، یا در خانه‌ی خود باقی بماند، یا به سـمت چپ حرکت کند یا به سـمت پایین یعنی خانه‌ی (۱۲, ۷) حرکت کند. الگوریتم Minimax سـودمندی پکمن را در جابجایی پکمن از خانه‌ی (۱۲, ۸) به خانه‌ی (۱۲, ۷) می‌بیند. بنابراین پکمن در حرکت بعدی خود به این خانه می‌رود. سپس دوباره بر اساس الگوریتم Minimax در صورتی که پکمن هنوز فاصـله‌ی مناسـبی با روح‌ها داشـته باشـد، طوری حرکات بعدیش را تنظیم می‌کند که در نهایت به نزدیک‌ترین غذا یعنی خانه‌ی (۱۱, ۶) برسد. شکل ۳-۲ تصویری از پکمن بعد از طی سه مرحله حرکت نشان می‌دهد که به کمک الگوریتم Minimax به موقعیت نزدیک‌ترین غذا رسیده است.



شکل ۳-۲- پکمن در خانه‌ی (۱۱, ۶)

حال دوباره با کمک الگوریتم Minimax پکمن باید تصـمیم بگیرد که به کدام سـمت حرکت کند. در این حالت تنها یک غذای دیگر باقی ملنده اسـت و آن در خلنه‌ی (۲ ,۹) قرار دارد. بنابراین پکمن بدون حرکت اضـافی و غیر سـودآور، و با توجه به فاصـله‌اش از روح‌ها باید تصـمیم بگیرد به گونه‌ای حرکت کند که بتواند به آخرین غذا برسد. پس در هر مرحله، با توجه به موقعیت جدید روح‌ها، پکمن تصمیم می‌گیرد

به چه ســمتی حرکت کند که در کمترین حرکات ممکن به خانه (۹, ۲) برســد. در این بازی با توجه به حرکت تصــادفی روح‌ها، الگوریتم Minimax برای پکمن تعیین می‌کند که در هر مرحله به ترتیب به خانه‌های (۱۱, ۵)، (۱۲, ۵)، (۱۲, ۴)، (۱۲, ۳)، (۱۱, ۳)، (۱۱, ۲)، (۱۰, ۲) و در نهایت خانه (۹, ۲) منتقل شـود و سـپس پکمن با رسـیدن به خانه (۹, ۲) و خوردن آخرین غذا، بازی را می‌برد. شـکل ۳-۳ تصـویر حالت پایانی این بازی را نمایش می‌دهد. قابل توجه است که اگر حرکت تصادفی روح‌ها به گونه‌ای بود که الگوریتم Minimax تشـخیص می‌داد در حرکات بعدی پکمن، احتمال برخورد او با یک روح وجود دارد، حرکات پکمن به گونه‌ای دیگر تغییر می‌کرد، برای مثال اگر در مرحله‌ای که پکمن در خانه‌ی (۳, ۱۲) قرار داشــت، یک روح در خانه‌ی (۳, ۱۱) قرار می‌گرفت، دیگر پکمن به خانه (۳, ۱۱) حرکت نمی‌کرد که به روح برخورد کند، بلکه بسته به محاسبه‌ی سودمندی او توسط الگوریتم Minimax، ممکن بود حرکت به خانه‌ی (۲, ۱۲) برای او حرکت بهتری باشــد و ســپس با توجه به حرکات بعدی روح‌ها، پکمن دوباره مسیر خود را پیدا و برنامه‌ریزی می‌کرد.



شکل ۳-۳- پکمن در خانه (۹, ۲)

۳-۲- عمق درخت بازی

برای آن که بتوانیم عمق مناسبی برای درخت بازی و الگوریتم Minimax انتخاب کنیم، یک راه آزمون و خطا توسط مقادیر مختلف برای عمق درخت است. در جدول ۳-۱ که جزئیات بیشتر آن در این لینک آورده شده است، نتایج بازی توسط ۶ عمق مختلف و برای هر عمق ده بار آزمایش، تحلیل شده است.

جدول۳-۱- نتایج ده بار آزمایش بازی با عمق‌های مختلف

عمق	میانگین امتیاز	بیشترین امتیاز	کمترین امتیاز	تعداد برد (از ۱۰ آزمایش)	میانگین تعداد مراحل	میانگین زمان اجرا (ثانیه)
۱	۷۱۱.۳	۹۴۳	۳۴۸	۵	۲۳۰.۷	۰.۱۵۸۲۰۰۸
۲	۷۵۹.۸	۹۴۴	۳۲۲	۶	۱۷۲.۲	۰.۲۰۰۹۸۹۴
۳	۸۴۴.۱	۹۷۲	۴۳۷	۱۰	۲۶۵.۹	۰.۶۲۷۰۶۱۲
۴	۹۲۸.۷	۹۷۳	۷۵۲	۱۰	۱۷۹.۳	۱.۵۱۸۷۰۷۶
۵	۹۰۷.۸	۹۶۳	۸۰۱	۱۰	۲۰۲.۲	۵.۶۱۶۸۰۲۶
۶	۹۱۶	۹۵۲	۸۶۶	۱۰	۱۹۴	۱۹.۲۴۰۲۴۳۹

بدون در نظر گرفتن نتایج آورده شده در این جدول نیز می‌توان حدس زد که عمق ۱ یا ۲ نمی‌تواند عمق مطلوبی برای بازی باشد، زیرا در این صورت پکمن تنها حداکثر تا دو حرکت بعدی خود را بتواند پیش بینی کند. طبق نتایج آزمایش‌ها، در عمق ۱ و ۲ احتمال باخت پکمن نیز وجود دارد، در حالی که در عمق‌های بیشتر این احتمال کمتر می‌شود و در این ده آزمایش در عمق‌های بیشتر پکمن هیچ یک را نباخته است. درست است که هر چقدر عمق بیشتر شود ممکن است بازی پکمن هم بهینه‌تر شود و نتایج بهتری به دست بیاید، اما زمان و حافظه‌ای که به کار برده می‌شود نیز بیشتر می‌شود و برای ما چنین چیزی مطلوب نیست. بنابراین با توجه به نتایج جدول ۳-۱، به نظر می‌رسد عمق ۴ عمق مطلوبی برای درخت بازی ما باشد؛ چرا که هم میانگین امتیاز آزمایش‌ها با این عمق نزدیک به میانگین امتیاز آزمایش‌ها با عمق‌های ۵ و ۶ است و این امتیازها بسیار بالاست، و هم اینکه زمان بسیار کمتری را نسبت به این دو عمق مصرف می‌کند. همچنین نتایج آزمایشات با عمق ۴، از نتایج آزمایشات با عمق ۳ بهتر است، با اینکه عمق ۳ زمان کمتری را مصرف می‌کند، اما میانگین امتیاز آن از میانگین امتیاز عمق ۴

کمتر اســـت و همچنین بیشـــترین و کمترین امتیازها در عمق ۳، بســـیار کمتر از بیشـــترین و کمترین امتیازها در عمق ۴ هســـتند. بنابراین ما در این بازی عمق ۴ را به عنوان عمقی مطلوب برای الگوریتم Minimax در نظر می‌گیریم و پکمن در هر مرحله می‌تواند تا چهار حرکت بعد از خود را پیش بینی کند.

فصل چهارم

لینک کد بازی Pac-Man

لینک کد بازی Pac-Man

همان طور که گفته شد، این بازی در محیط pycharm برنامه‌نویسی شده است، و برای به اشتراک گذاشتن با دیگران، این کد در Google Colab نیز بارگزاری شده و لینک آن در زیر قابل دسترسی است:

لینک کد بازی Pac-Man در Google Colab

فصل پنجم

جمع‌بندی و نتیجه‌گیری

جمع‌بندی و نتیجه‌گیری

در این مقاله به تحلیل و بررسی کد بازی Pac-Man پرداختیم که در آن الگوریتم Minimax پیاده‌سازی شده است. الگوریتم Minimax در بهینه بازی کردن عامل پکمن تاثیر بسیار زیادی دارد و با توجه به تعریف تابع e-utility و کاربرد آن در درخت بازی، این الگوریتم تصمیم می‌گیرد که کدام حرکت پکمن برای او سودمندی بیشتری دارد و می‌تواند بدون برخورد به روح‌ها و در کمترین حرکات ممکن تمام غذاهای موجود بر روی صفحه بازی را بخورد. دیدیم که هر چقدر تابع e-utility بهتر تعریف شود، به وضعیت‌های غیر پایانی بازی امتیاز دقیق‌تری نسبت داده می‌شود و در نتیجه کارآیی الگوریتم Minimax افزایش پیدا می‌کند. همچنین عمق درخت بازی در این کارآیی نقش مهمی را ایفا می‌کند؛ درست است که عمق بیشتر می‌تواند نتایج بهتری را به دنبال داشته باشد، اما از طرف دیگر نیازمند حافظه و زمان بیشتری است و ما با توجه به تحلیل چندین آزمایش با عمق‌های مختلف در بازی، عمق مطلوب خود را پیدا کردیم.

منابع

سهراب جلوه‌گر، "کتاب الکترونیکی هوش مصنوعی"، ویرایش ۲۳، سال ۹۸

"Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Minimax.

"Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Game_tree.

Stuart Russell, Peter Norvig, "Artificial Intelligence, A Modern Approach", Fourth Edition, 2021