مدرس: دکتر فاطمه منصوری

نیم‌سال اول ۱۴۰۲-۱۴۰۳

تهیه کننده: فاطمه حسین زاده

هوش مصنوعی

جزوه پنجم

بازی ها:

در یادداشت اول، ما در مورد مسائل جستجو و نحوه حل آنها به طور کارآمد و بهینه صحبت کردیم - با استفاده از الگوریتم های قدرتمند جستجوی تعمیم یافته، عوامل ما می توانند بهترین طرح ممکن را تعیین کنند و سپس به سادگی آن را برای رسیدن به یک هدف اجرا کنند . حال، بیایید دنده‌ها را تغییر دهیم و سناریوهایی را در نظر بگیریم که در آن عوامل ما یک یا چند دشمن[1] دارند که سعی می‌کنند آنها را از رسیدن به اهدافشان باز دارند .عوامل ما دیگر نمی‌توانند الگوریتم‌های جستجویی را که قبلاً یاد گرفته‌ایم برای تدوین یک برنامه اجرا کنند، زیرا ما معمولاً نمی‌دانیم که دشمنان ما چگونه علیه ما برنامه‌ریزی می‌کنند و به اقدامات ما پاسخ می‌دهند .در عوض، ما نیاز به اجرای یک کلاس جدید از الگوریتم‌ها داریم که راه‌حل‌هایی را برای مسائل جستجوی خصمانه[2] ارائه می‌دهند که معمولاً به عنوان بازی شناخته می‌شوند.

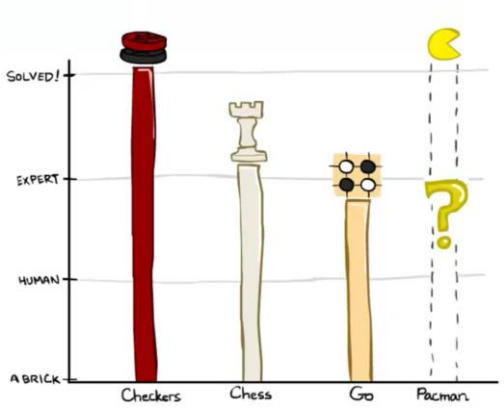
انواع مختلفی از بازی ها وجود دارد .بازی ها می توانند کنش هایی با نتایج قطعی یا تصادفی (احتمالی) داشته باشند، یا می توانند تعداد بازیکنان متغیری داشته باشند و یا ممکن است مجموع صفر[3] باشند یا نباشند .اولین دسته از بازی‌هایی که به آن خواهیم پرداخت، بازی‌های مجموع صفر قطعی[4] هستند، بازی‌هایی که در آنها اقدامات قطعی هستند و سود ما مستقیماً معادل باخت حریف است و بالعکس .ساده ترین راه برای فکر کردن در مورد چنین بازی هایی این است که با یک مقدار متغیر تعریف شوند، که یک تیم یا عامل سعی می کند آن را به حداکثر برساند و تیم یا عامل مقابل سعی می کند آن را به حداقل برساند و آنها را در رقابت مستقیم قرار دهد .در پک من، این متغیر امتیاز شما است که سعی می‌کنید با خوردن سریع و کارآمد گلوله‌ها آن را به حداکثر برسانید در حالی که ارواح سعی می‌کنند ابتدا شما را بخورند . بسیاری از بازی های خانگی رایج نیز در این دسته از بازی ها قرار می گیرند:

- چکرز:[5] اولین بازیکن کامپیوتری چکرز در سال ۱۹۵۰ ساخته شد. از آن زمان، چکرز به یک بازی حل شده تبدیل شده است، به این معنی که هر موقعیتی را می توان به عنوان یک برد، باخت یا مساوی برای هر دو طرف ارزیابی کرد، زیرا هر دو بازیکن به طور بهینه عمل می کنند.
- شطرنج: در سال ۱۹۹۷، دیپ بلو[6] اولین عامل کامپیوتری شد که قهرمان شطرنج انسانی گری کاسپاروف را در یک مسابقه شش بازی ای شکست داد. دیپ بلو برای استفاده از روش های بسیار پیچیده برای ارزیابی بیش از ۲۰۰ میلیون موقعیت در ثانیه ساخته شده است .برنامه های فعلی حتی بهترند، هرچند کمتر تاریخی اند.

---

[1] adversaries
[2] adversarial search problems
[3] zero-sum
[4] deterministic zero-sum games
[5] Checkers
[6] Deep Blue

- گو:[7] فضای جستجو برای این بازی بسیار بزرگتر از شطرنج است، و بنابراین اکثر آنها باور نداشتند که عاملان کامپیوتری هرگز قهرمانان جهان را برای چندین سال آینده شکست دهند .با این حال، AlphaGo، توسعه یافته توسط گوگل، قهرمان Go Lee Sodolرا در مارس ۲۰۱۶، ۴ بر ۱ شکست داد.



همه عوامل قهرمان جهان که در بالا گفتیم، حداقل تا حدی، از تکنیک‌های جستجوی خصمانه استفاده می‌کنند که می‌خواهیم درباره آنها صحبت کنیم .برخلاف جستجوی معمولی که یک طرح جامع را برمی‌گرداند، جستجوی خصمانه یک استراتژی یا خط مشی را برمی‌گرداند که با توجه به پیکربندی‌های عامل(های) ما و مخالفان آنها، به سادگی بهترین حرکت ممکن را توصیه می‌کند .به زودی خواهیم دید که چنین الگوریتم‌هایی دارای ویژگی زیبایی هستند که از طریق محاسبات باعث ایجاد رفتار می‌شوند - محاسباتی که ما اجرا می‌کنیم از نظر مفهومی نسبتاً ساده و به طور گسترده قابل تعمیم است، با این حال به طور ذاتی یک همکاری بین عوامل در همان تیم و همچنین "تفکر" در مورد عوامل متخاصم ایجاد می‌کنند.

فرمول استاندارد بازی شامل تعاریف زیر است:

- حالت شروع: s_0
- بازیکنان: بازیکن (بازیکنان) مشخص می کنند که نوبت چه کسی است
- کنش ها: کنش(کنش ها) ی قابل اجرا برای هر بازیکن
- نتیجه مدل انتقال(s,a)
- بررسی پایانه: Terminal-test(s)
- مقادیر پایانه: Utility(s,player)

مینیماکس(Minimax):

[استراتژی «حداقل-حداکثر» یا «کمین‌بیش»، یک استراتژی محتاطانه است که حداکثر ضرر را به حداقل می‌رساند. این استراتژی برای یک تصمیم گیرنده محافظه کار مفید است. اساساً این روش برای فرار از باخت و حداقل کردن ضررها صورت گرفته و مخصوص افرادی است که که نمی‌خواهند تصمیم اشتباه بگیرند.

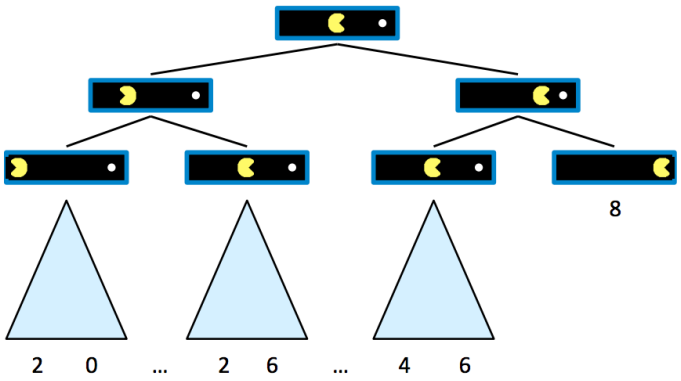
[7] Go

این الگوریتم تصمیم مینیماکس را از حالت فعلی محاسبه میکند و مقادیر مینیماکس مربوط به هر حالت پسین را بطور بازگشتی با پیاده سازی مستقیم معادلات تعریف شده، محاسبه میکند. این محاسبات بازگشتی به طرف پایین تا برگهای درخت پیش میرود و سپس در برگشت از حالت بازگشتی، مقادیر مینیماکس هر گره را اختصاص میدهد.

الگوریتم مینیماکس اکتشاف عمقی کاملی را روی درخت بازی انجام میدهد. اگر حداکثر عمق درخت m باشد و در هر نقطه b حرکت معتبر وجود داشته باشد، آنگاه پیچیدگی زمانی الگوریتم مینیماکس برابر است با $O(b^m)$. برای الگوریتمی که تمام فعالیت ها را همزمان تولید میکند، پیچیدگی فضا (حافظه) برابر با O(bm)، یا برای الگوریتمی که هر بار یک فعالیت را تولید میکند، برابر با O(m) است. البته برای بازی های واقعی هزینه زمان کاملا غیرعملی است. اما این الگوریتم به عنوان مبنایی برای تحلیل ریاضی بازی ها و برای بسیاری از الگوریتم های عملی است.]

اولین الگوریتم بازی مجموع صفر که در نظر خواهیم گرفت، مینیماکس است، که با این فرض انگیزشی اجرا می شود که حریفی که با آن روبرو هستیم رفتار بهینه ای دارد و همیشه حرکتی را که برای ما بدترین حالت است را انجام می دهد .برای معرفی این الگوریتم، ابتدا باید مفهوم مطیوبیت های پایانه[8]و مقدار حالت[9]را شرح دهیم .مقدار یک حالت امتیاز بهینه ای است که توسط عاملی که آن حالت را کنترل می کند به دست می آید .برای درک معنای این موضوع، صفحه بازی پک من ساده زیر را مشاهده کنید:

فرض کنید که پک من با ۱۰ امتیاز شروع می کند و در هر حرکت ۱ امتیاز از دست می دهد تا زمانی که گلوله را بخورد، در این مرحله بازی به حالت پایانی می رسد و بازی به پایان می رسد .می‌توانیم به شکل زیر شروع به ساخت درخت بازی برای این بازی کنیم، که در آن فرزندان یک حالت، وضعیت‌های جانشین[10]هستند، درست مانند درخت‌های جستجو برای مسائل جستجوی عادی:



از این درخت مشخص است که اگر پکمن مستقیماً به سمت گلوله برود، بازی را با امتیاز ۸ به پایان می‌رساند، در حالی که اگر در هر نقطه عقب نشینی کند، در نهایت با مقداری با امتیاز کمتر به پایان می‌رسد .اکنون که یک درخت بازی با چندین حالت پایانه و واسطه ایجاد کرده‌ایم، آماده هستیم تا معنای ارزش هر یک از این حالت‌ها را شرح دهیم.

ارزش یک حالت به عنوان بهترین نتیجه ممکن (مطلوب) تعریف می شود که یک عامل می تواند از آن حالت به دست آورد .ما مفهوم مطلوبیت را بعداً به طور ملموس‌تر شرح می‌دهیم، اما در حال حاضر کافی است به سود یک عامل به عنوان امتیاز یا تعداد امتیازهایی که به دست

[8] terminal utilities
[9] state value
[1] successor states

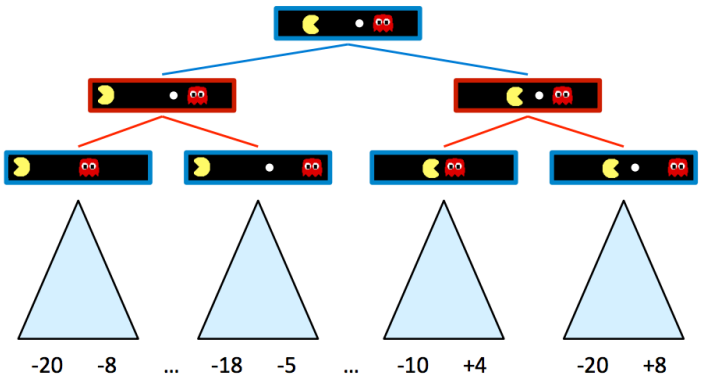
می‌آورد فکر کنیم .مقدار یک حالت پایانه، که مطلوبیت پایانه[1] نامیده می شود، همیشه مقداری از مقادیر شناخته شده قطعی و یک ویژگی ذاتی از بازی است .در مثال پک من ما، مقدار سمت راست ترین حالت پایانه به سادگی ۸ است، امتیازی که پک من با رفتن مستقیم به سمت گلوله به دست می آورد .همچنین در این مثال مقدار یک حالت غیر پایانی به عنوان حداکثر مقادیر فرزندان آن تعریف شده است .با تعریف V(s)به عنوان تابعی که مقدار یک حالت s را تعریف می کند، می توانیم مبحث فوق را خلاصه کنیم:

$$\forall \text{non-terminal states}, \quad V(s) = \max_{s' \in successors(s)} V(s')$$
$$\forall \text{terminal states}, \quad V(s) = \text{known}$$

این یک قاعده بازگشتی بسیار ساده را تنظیم می کند، که طبق آن منطقی است که مقدار فرزند راست مستقیم گره ریشه ۸ باشد و فرزند سمت چپ مستقیم گره ریشه ۶ باشد، زیرا اینها حداکثر امتیازات احتمالی هستند که عامل می تواند به دست آورد، اگر به ترتیب از حالت شروع به سمت راست یا چپ حرکت کند .نتیجه این است که با اجرای چنین محاسباتی، یک عامل می تواند بفهمد که حرکت به راست بهینه است، زیرا فرزند سمت راست ارزش بیشتری نسبت به فرزند سمت چپ از حالت شروع دارد.

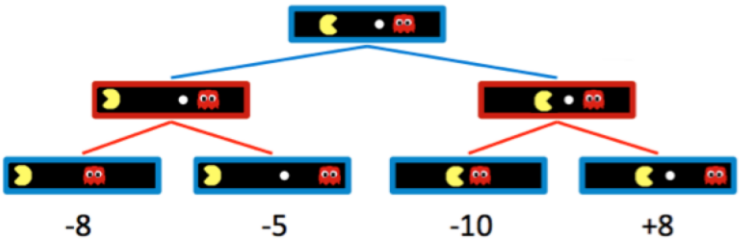
حالا بیایید یک صفحه بازی جدید با یک روح خصمانه معرفی کنیم که می‌خواهد جلوی پک من را از خوردن گلوله بگیرد.

طبق قوانین بازی دو عامل به نوبت حرکت می کنند و به درخت بازی ای منتهی می شوند که در آن دو عامل لایه های درخت را که "تحت کنترل دارند" عوض می کنند.کنترل یک عامل بر روی یک گره ، یعنی گره با حالتی مطابقت دارد که در آن نوبت آن عامل است، و بنابراین این فرصت برای اوست که در مورد یک عمل تصمیم بگیرد و وضعیت بازی را بر اساس آن تغییر دهد .در اینجا درخت بازی حاصل از صفحه جدید بازی دو عاملی بالا را داریم:



گره های آبی مربوط به گره های تحت کنترل پک من است و در آنها پک من می تواند تصمیم بگیرد که چه اقدامی انجام دهد، در حالی که گره های قرمز مربوط به گره های تحت کنترل روح است .توجه کنید که همه فرزندان گره‌های تحت کنترل روح، گره‌هایی هستند که روح از حالت قبلی خود در والدش به چپ یا راست حرکت کرده است، و همینطور این روند برای گره‌های تحت کنترل پک من نیز رخ داده است. برای سادگی، بیایید این درخت بازی را به درخت عمق ۲ تقلیل بذهیم و مقادیر جعلی را به حالت های پایانه ها به صورت زیر اختصاص دهیم:

[1] terminal utility



طبیعتاً، افزودن گره‌های تحت کنترل روح، حرکتی را که پک من معتقد است بهینه است را، تغییر می‌دهد و حرکت بهینه جدید با الگوریتم مینیماکس تعیین می‌شود .الگوریتم مینیماکس به جای به حداکثر رساندن سودمندی بر روی فرزندان در هر لایه از درخت، فقط فرزندان گره های کنترل شده توسط پک من را به حداکثر می رساند، در حالی که فرزندان گره های تحت کنترل روح را به حداقل می رساند .از این رو، دو گره روح بالا به ترتیب دارای مقادیر min(-8,-5) = -8 و min (-10,+8) = -10 هستند .به همین ترتیب، گره ریشه نحن کنترل پک من دارای مقدار max(-8,-10) = -8 است .از آنجایی که پکمن می‌خواهد امتیاز خود را به حداکثر برساند، به سمت چپ می‌رود و امتیاز ۸- را می‌گیرد به جای اینکه به دنبال گلوله برود و امتیاز ۱۰- را بگیرد .این نمونه بارز تغییر رفتار از طریق محاسبات است - اگرچه پکمن می‌خواهد امتیاز ۸+ را دریافت کند، ولی اگر در نهایت در سمت راست‌ترین حالت فرزند قرار بگیرد، از طریق مینیماکس «می‌داند» که یک روح با عملکرد بهینه به او اجازه نمی‌دهد آن امتیاز را کسب کند .به منظور انجام بهینه، پک من مجبور می‌شود شرط‌های خود را رعایت کند و به طور غیرمنتظره از گلوله دور شود تا کاهش امتیاز خود را به حداقل برساند .ما می توانیم روشی که مینیماکس مقادیر را به حالت ها اختصاص می دهد به صورت زیر خلاصه کنیم:

$$\begin{aligned}
\forall \text{agent-controlled states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\
\forall \text{opponent-controlled states}, \quad V(s) &= \min_{s' \in successors(s)} V(s') \\
\forall \text{terminal states}, \quad V(s) &= \text{known}
\end{aligned}$$

در پیاده‌سازی، مینیماکس مانند جستجوی اول عمق عمل می‌کند و مقادیر گره‌ها را به همان ترتیبی که DFS انجام می‌دهد محاسبه می‌کند، از سمت چپ‌ترین گره شروع می کند و به طور مکرر به سمت راست حرکت می‌کند .به طور دقیق تر، پیمایش پس ترتیب[1] از درخت بازی را انجام می دهد .شبه کد به دست آمده برای مینیماکس هم زیبا و هم از نظر شهودی ساده است و در زیر ارائه شده است .توجه داشته باشید که مینیماکس یک عمل را که مربوط به گره ریشه است را به فرزندی که مقدار خود را از آن گرفته است، برمی گرداند.

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the agent is MAX: return max-value(state)
    if the agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

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

[1] postorder traversal

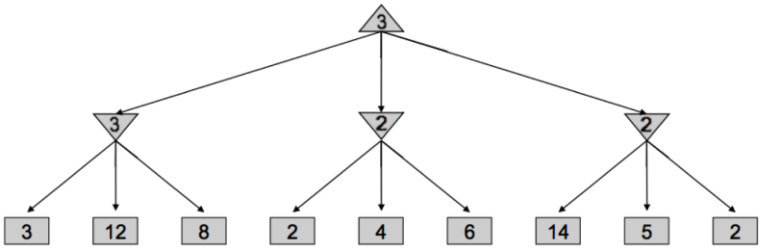
هرس آلفا-بتا:

[هرس آلفا-بتا (alpha-beta pruning) روشی است برای یافتن جواب بهینه‌ی الگوریتم مینیماکس که از جست‌وجو در برخی فضاهایی که انتخاب نمی‌شوند، جلوگیری می‌کند. به این ترتیب جست‌وجو در درخت کمینه-بیشینه تا عمق مشخصی در زمان کمتری انجام می‌شود.

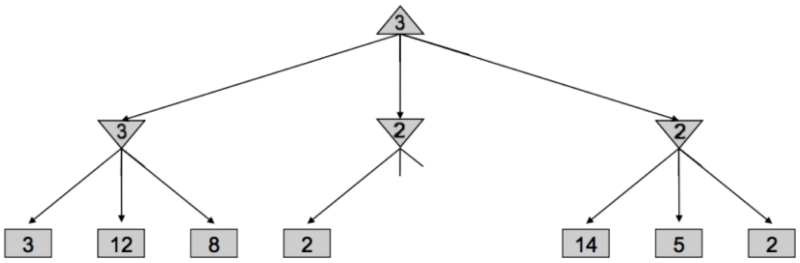
این الگوریتمی کارایی الگوریتم درخت مینیماکس (درخت کمینه بیشینه یا درخت بازی) را بهبود می‌بخشد. با استفاده از هرس آلفا بتا، بخش‌هایی از درخت کمینه بیشینه که پیمایششان بی تأثیر است پیمایش نمی‌شوند و به این ترتیب پیمایش درخت کمینه بیشینه تا یک عمق مشخص در زمانی کم‌تر صورت می‌گیرد.]

مینیماکس تقریباً کامل به نظر می رسد چراکه ساده، بهینه و شهودی است. با این حال، اجرای آن بسیار شبیه به جستجوی اول عمق است و پیچیدگی زمانی اش با آن یکسان است، یعنی $O(b^m)$. با یادآوری اینکه b عامل انشعاب و m عمق تقریبی درختی است که گره های پایانه را می توان در آن یافت، این الگوریتم زمان اجرای بسیار زیادی را برای بسیاری از بازی ها به همراه دارد. برای مثال، شطرنج با ضریب انشعاب $b \approx 35$ و عمق درخت $m \approx 100$ است پس پیچیدگی زمانی اش خیلی زیاد میشود. برای کاهش این مقدار، مینیماکس یک بهینه سازی دارد که هرس آلفا-بتا است.

از نظر مفهومی، هرس آلفا-بتا به این صورت است: اگر می‌خواهید ارزش گره n را با نگاه کردن به جانشین‌های آن تعیین کنید، به محض اینکه متوجه شدید که مقدار n می‌تواند در بهترین حالت با مقدار بهینه والد n برابری کند، دیگر به دنبال آن نگردید .با یک مثال معنای این جمله را شرح می دهیم .درخت بازی زیر را با گره‌های مربعی مربوط به حالت‌های پایانی، مثلث‌های رو به پایین مربوط به گره‌های کمینه سازی، و مثلث‌های رو به بالا مربوط به گره‌های بیشینه سازی را در نظر بگیرید:



بیایید نحوه استخراج این درخت را با الگوریتم مینیماکس بررسی کنیم، با تکرار در گره‌ها با مقادیر ۳، ۱۲، و ۸، و اختصاص مقدار 3،min(3،12،8) = 3 به سمت چپ ترین کمینه ساز شروع می‌کنیم. سپس، min(2،4،6) =2را به کمینه سازمیانی، و min(14، 5، 2) =2را به سمت راست ترین کمینه ساز، قبل از اینکه در نهایت max(3، 2، 2) =3را به بیشینه ساز واقع در ریشه اختصاص دهیم، اختصاص می‌دهیم . با این حال، اگر به این وضعیت فکر کنیم، می‌توانیم به این موضوع پی ببریم که به محض بازدید از فرزند کمینه ساز وسط با مقدار ۲، دیگر نیازی نیست به سایر فرزندان کمینه ساز وسط نگاه کنیم. چراکه ما فرزندی از کمینه ساز وسط با مقدار ۲ دیده‌ایم، می‌دانیم که مهم نیست فرزندان دیگر چه مقادیری دارند، چون مقدار کمینه ساز وسط حداکثر می‌تواند ۲ باشد. اکنون که این مورد مشخص شد، بیایید یک گام بعدی را بررسی کنیم .هنوز هم بیشینه ساز در ریشه باید از بین مقدار ۳ از کمینه ساز سمت چپ و مقدار $\geq$ ۲ انتخاب کند، تضمین می شود که بیشینه ساز مقدار ۳ که توسط کمینه ساز چپ برگردانده شده است را بر مقدار بازگشتی کمینه ساز وسط ، بدون توجه به مقادیر فرزندان باقی مانده، ترجیح می دهد. دقیقاً به همین دلیل است که می‌توانیم درخت جستجو را هرس کنیم و هرگز فرزندان باقی‌مانده کمینه ساز وسط را بررسی نکنیم:



اجرای چنین هرسی می تواند زمان اجرا را به $O(b^{m/2})$ کاهش دهد و به طور موثر عمق "قابل حل" ما را دو برابر کند .در عمل، اغلب بسیار کمتر از این مقدار است، اما به طور کلی می تواند جستجو در حداقل یک یا دو سطح دیگر را امکان پذیر کند .که این نکته قابل توجهی است، چراکه بازیکنی که فکر می کند ۳ حرکت جلوتر است، بر بازیکنی که فکر می کند ۲ حرکت جلوتر است برای برد ترجیح داده می شود .این هرس دقیقاً همان کاری است که الگوریتم مینیماکس با هرس آلفا-بتا انجام می دهد و به صورت زیر پیاده سازی می شود:

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
```

این کد را با کد اصلی مینیملکس مقایسه کنید، و توجه داشته باشید که اکنون می‌توانیم زودتر بدون جست‌وجو در میان همه جانشین‌ها به نتیجه برسیم.

توابع ارزیابی:

اگرچه هرس آلفا-بتا می‌تواند به افزایش عمقی که می‌توانیم اجرا کنیم، کمک می‌کند، اما معمولاً به اندازه کافی برای رسیدن به انتهای درخت‌های جستجو برای اکثر بازی‌ها خوب نیست .در نتیجه، ما به توابع ارزیابی روی می‌آوریم، توابعی که یک حالت را می‌گیرند و تخمینی از مقدار کمینه واقعی آن گره را به دست می‌آورند .به طور معمول، در این توابع حالت هایی به عنوان حالت های "بهتر" تعبیر می شوند که مقادیر بالاتری را در مقابل حالت های "بدتر" نسبت می دهند .توابع ارزیابی به‌طور گسترده در مینیماکس های محدود به عمق استفاده می‌شوند، جایی که گره‌های غیر پایانی واقع در حداکثر عمق قابل حل خود را به عنوان گره‌های پایانه در نظر می‌گیریم، و به آنها ابزار پایانه ساختگی می‌دهیم که توسط یک تابع ارزیابی با دقت انتخاب شده تعیین می‌شود .از آنجایی که توابع ارزیابی فقط می توانند تخمین هایی از مقادیر ابزارهای غیر پایانی ارائه دهند، تضمین بهینه بودن بازی را هنگام اجرای مینیماکس حذف می کنند.

هنگام طراحی عاملی که مینیماکس را اجرا می کند، معمولاً برای انتخاب یک تابع ارزیابی، فکر و آزمایش های زیادی انجام می شود و هر چه عملکرد تابع ارزیابی بهتر باشد، عامل به رفتار بهینه نزدیکتر می شود .علاوه بر این، قبل از استفاده از یک تابع ارزیابی، عمیق‌تر شدن در درخت نیز نتایج بهتری به ما می‌دهد، زیرا انجام محاسبات آنها درعمق درخت بازی، به خطر افتادن بهینه‌گی را کاهش می‌دهد . هدف این توابع در بازی ها بسیار مشابه توابع اکتشافی در مسائل جستجوی استاندارد است.

رایج ترین طراحی برای یک تابع ارزیابی، ترکیب خطی از ویژگی ها است.

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

هر $f_i(s)$ مربوط به یک ویژگی استخراج شده از حالت ورودی s است و به هر ویژگی یک وزن w_i اختصاص داده می شود .ویژگی‌ها به سادگی برخی از عناصر یک حالت بازی هستند که می‌توانیم آن‌ها را استخراج و یک مقدار عددی به آنها اختصاص دهیم .به عنوان مثال، در یک بازی چکرز ممکن است یک تابع ارزیابی با ۴ ویژگی بسازیم :تعداد مهره های[3]عامل، تعداد شاه‌های عامل، تعداد مهره های حریف و تعداد شاه‌های حریف .سپس وزن‌های مناسب را بر اساس اهمیت آن‌ها انتخاب می‌کنیم .در مثال چکرز، انتخاب وزن های مثبت برای مهره ها/پادشاهان عامل خود و وزن های منفی برای مهره ها/پادشاهان حریف بسیار منطقی است .علاوه بر این، ممکن است تصمیم بگیریم که به علت ارزش بیشتر پادشاهان نسبت به بقیه مهره‌ها ، به ویژگی‌های مربوط به پادشاهان عامل/رقیب وزن‌های بیشتری نسبت به ویژگی‌های مربوط به بقیه مهره‌ها بدهیم .در زیر یک تابع ارزیابی احتمالی داریم که با ویژگی‌ها و وزن‌هایی که اخیراً بررسی کردیم مطابقت دارد:

$$Eval(s) = 2 \cdot agent\_kings(s) + agent\_pawns(s) - 2 \cdot opponent\_kings(s) - opponent\_pawns(s)$$

همانطور که مشخص است، طراحی تابع ارزیابی می تواند کاملاً آزاد باشد و لزوماً نباید از توابع خطی باشد .برای مثال، توابع ارزیابی غیرخطی مبتنی بر شبکه های عصبی در برنامه های یادگیری تقویتی[4]بسیار رایج هستند .مهم ترین چیزی که باید در نظر داشت این است که تابع ارزیابی تا جایی که ممکن است، باید امتیازات بالاتری را برای موقعیت های بهتر به دست بیاورد .این ممکن است نیاز به تنظیم دقیق و آزمایش های متعدد روی عملکرد عوامل با استفاده از توابع ارزیابی با ویژگی‌ها و وزن‌های مختلف داشته باشد.

---

[1] pawns

[1] Reinforcement Learning applications

[3]

[4]