|  |  |  |
| --- | --- | --- |
|  | **با نام و یاد او** | |
|  | |
|  | |
| **دانشکده مهندسی و علوم کامپیوتر**  **گزارش فاز اول پروژه هوش مصنوعی و سیستم های خبره – دکتر سلیمی‌بدر**  **پاییز ۱۴۰۳** | |  |

**گردآورندگان: آراس ولی‌زاده – امیرسامان رستم‌بیگی**

داخل این کد یه عامل هوشمند برای بازی “Hand of the King” طراحی کردیم که از الگوریتم **مینیمکس با هرس آلفا-بتا** استفاده می‌کنه تا بهترین حرکت رو برای بازیکن پیدا کنه.

**عامل هوشمند چیه؟**

عامل هوشمند یه برنامه یا کد کامپیوتریه که می‌تونه وضعیت بازی رو تحلیل کنه و بر اساس استراتژی‌های تعریف‌شده، بهترین حرکت ممکن رو انتخاب کنه. اینجا عامل ما:

* حرکت‌های ممکن رو بررسی می‌کنه.
* امتیاز وضعیت‌های مختلف بازی رو می‌سنجه.
* حرکتی رو انتخاب می‌کنه که بیشترین احتمال برد رو داشته باشه.

**الگوریتم مینیمکس چیه؟**

الگوریتم مینیمکس یکی از روش‌های استاندارد برای حل مسائل بازی‌های نوبتیه، مثل شطرنج یا این بازی. توی این الگوریتم:

* **بازیکن ما** (عامل هوشمند) تلاش می‌کنه امتیاز رو ماکزیمم کنه.
* **حریف** تلاش می‌کنه امتیاز رو مینیمم کنه.
* مینیمکس این تعامل رو شبیه‌سازی می‌کنه و وضعیت‌هایی که هر بازیکن می‌تونه انتخاب کنه رو بررسی می‌کنه.

**هرس آلفا-بتا چیه؟**

هرس آلفا-بتا یه بهبود برای الگوریتم مینیمکسه که باعث می‌شه شاخه‌هایی که نتیجه‌ی قطعی دارن، بررسی نشن. این کار دو تا مزیت داره:

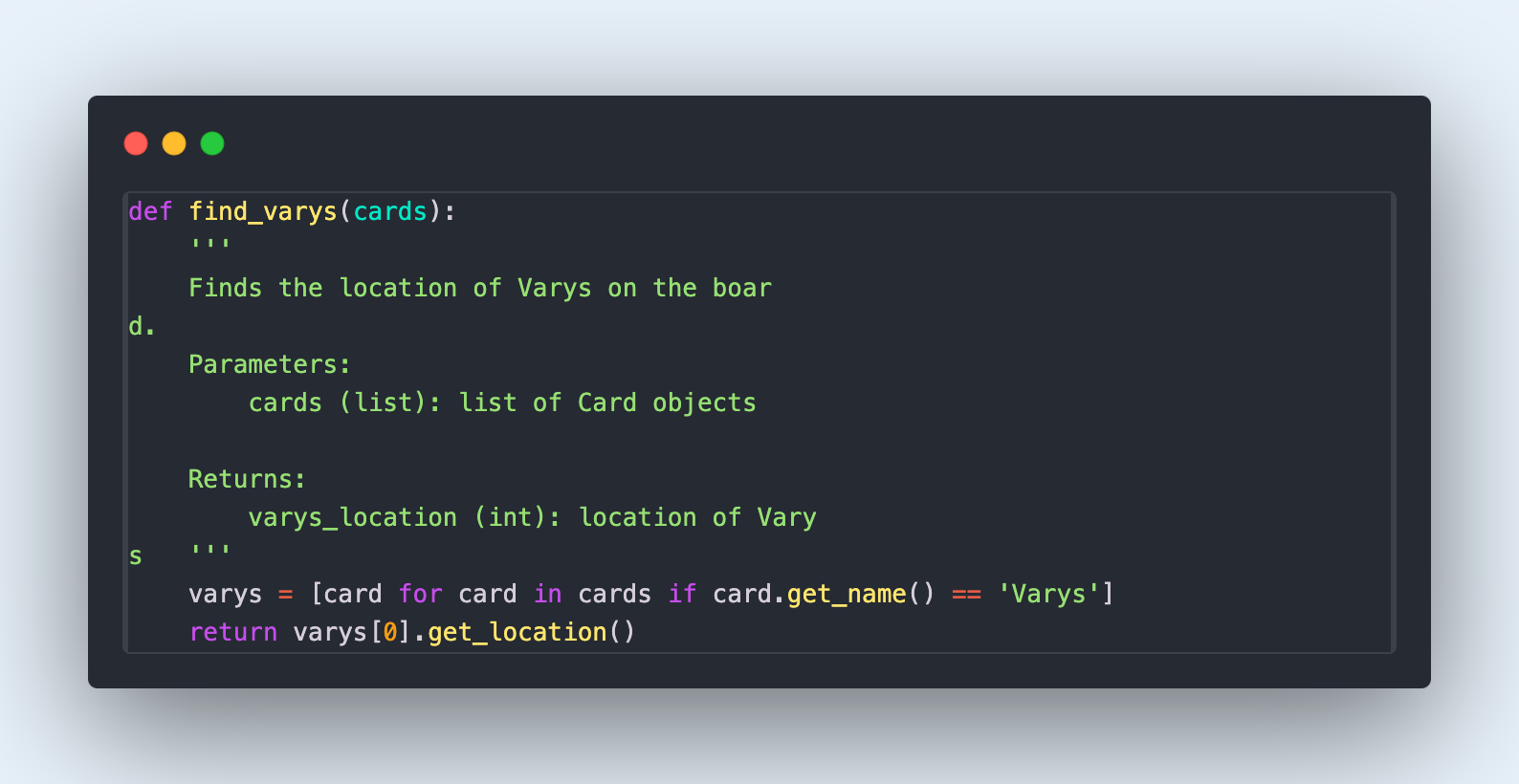
* **سرعت بیشتر**: شاخه‌های بی‌فایده حذف می‌شن.
* **بهره‌وری بالاتر**: فقط وضعیت‌هایی که اهمیت دارن بررسی می‌شن.

مثلاً اگه توی یه بازی وضعیت A امتیاز بهتری نسبت به وضعیت B داشته باشه، نیازی نیست بقیه حرکت‌های B رو بررسی کنیم.

**چرا مینیمکس اینجا خوب کار می‌کنه؟**

* بازی “Hand of the King” یه بازی نوبتیه که حرکت‌های ممکنش قابل پیش‌بینیه.
* عامل هوشمند می‌تونه حرکت‌های بعدی خودش و حریف رو شبیه‌سازی کنه.
* هدف اینه که **هر حرکتی که انتخاب می‌کنه، بیشترین امتیاز رو در آینده براش بیاره.**

حالا میتونیم بریم سراغ اینکه بخش های مختلف کد رو توضیح بدیم:



**پیدا کردن مکان واریس (تابع find\_varys)**

این تابع توی لیست کارت‌ها می‌گرده و کارت “Varys” رو پیدا می‌کنه. بعد مکانش رو (که یه عدده و نشون‌دهنده جای کارت توی برده) برمی‌گردونه. از این برای مشخص کردن حرکت‌های معتبر استفاده می‌کنیم.

\

****

**پیدا کردن حرکت‌های معتبر (تابع get\_valid\_moves)**

این تابع مشخص می‌کنه واریس تو کدوم جهت‌ها (راست، چپ، بالا یا پایین) می‌تونه حرکت کنه:

* مکان واریس رو می‌گیره.
* کارت‌هایی که توی همون ردیف یا ستون هستن رو پیدا می‌کنه.
* یه لیست از مکان‌های ممکن رو برمی‌گردونه.

****

**ارزیابی وضعیت بازی (تابع evaluate\_state)**

این تابع واقعا قلب تصمیم‌گیری ایجنت ماست و کل عملکردش به این وابسته‌ست که بازی رو چجوری ارزیابی کنه. هدفش اینه که به هر وضعیت بازی یه امتیاز بده که این امتیاز مشخص کنه بازی چقدر به نفع بازیکن اول (عامل هوشمند ما) هست.

**چطوری کار می‌کنه؟**

1. **امتیاز بنرها:**

وقتی یه بازیکن بتونه یه خاندان رو کامل کنه (یعنی کارت‌های لازم اون خاندان رو جمع کنه)، بنر اون خاندان رو می‌گیره. اینجا تابع به هر بنر یه امتیاز خاص می‌ده که بر اساس ارزش خاندانه. مثلا خاندان Stark امتیاز بالاتری داره چون تعداد کارت‌های بیشتری داره، در حالی که Tully با دو کارت تکمیل می‌شه ولی امتیاز کمتری داره.

• بازیکن اول بنر بگیره؟ امتیاز مثبت می‌گیره.

• بازیکن دوم بنر بگیره؟ امتیاز منفی برای بازیکن اول ثبت می‌شه.

2. **امتیاز کارت‌ها:**

اینجا تابع نگاه می‌کنه که هر بازیکن چندتا کارت از یه خاندان خاص داره.

• اگر کارت‌های بازیکن اول به تعداد کافی برسه که بنر رو بگیره، یه امتیاز اضافی می‌گیره.

• اگر تعداد کارت‌هاش کمتر از نصف کارت‌های اون خاندان باشه، امتیاز کمتری می‌گیره، چون تو رقابت ممکنه عقب بمونه.

3. **حرکت‌های ممکن:**

تابع تعداد حرکت‌های ممکن هر بازیکن رو هم چک می‌کنه. اگه بازیکن اول حرکت‌های بیشتری داشته باشه، امتیاز مثبت می‌گیره. ایده اینه که بازیکنی که گزینه‌های بیشتری داره، احتمالا کنترل بیشتری رو بازی داره.

**چرا مهمه؟**

این تابع باعث می‌شه ایجنت ما بدونه کدوم وضعیت بازی براش بهتره. وقتی الگوریتم Minimax می‌خواد حرکت‌های مختلف رو امتحان کنه، از این تابع برای مقایسه وضعیت‌ها استفاده می‌کنه. اگه این ارزیابی درست کار کنه، ایجنت می‌تونه تصمیم‌های بهتری بگیره.

**یه نکته باحال :)**

اینجا ما به بنرهای کامل‌شده وزن بیشتری دادیم چون هدف نهایی بازی همین تصاحب بنرهاست. اما تعداد کارت‌ها و تعداد حرکت‌ها هم تاثیر داره که ایجنت تو مسیر درست حرکت کنه و شانسی برای برد به حریف نده.

به زبون ساده: این تابع مثل یه متر کار می‌کنه که وضعیت بازی رو می‌سنجه و می‌گه کدوم بازیکن جلوتره و چقدر جلوئه.

****

**شبیه‌سازی حرکت (تابع simulate\_move)**

خب این تابع یه جورایی مثل یه ماشین زمان می‌مونه! وقتی یه حرکت رو انجام بدیم، این تابع بازی رو جلو می‌بره و وضعیت جدید رو بهمون نشون می‌ده، بدون اینکه واقعاً وضعیت اصلی بازی رو تغییر بده. از اینجا به بعد، ایجنت می‌تونه این وضعیت شبیه‌سازی‌شده رو بررسی کنه و ببینه این حرکت خوبه یا نه.

* اول از همه، وضعیت فعلی بازی شامل کارت‌ها، بازیکن‌ها و هر چیزی که تو بازی هست رو **کپی** می‌کنه. این کپی باعث می‌شه که وضعیت اصلی بازی دست‌نخورده بمونه.
* بعدش، حرکت موردنظر رو انجام می‌ده. این یعنی مثلا Varys رو از جایی به جای دیگه می‌بره و کارت مربوطه رو می‌گیره.
* توی این حرکت، ممکنه بازیکن یه خاندان رو کامل کنه. وقتی این اتفاق بیفته، پرچم (بنر) اون خاندان به بازیکن داده می‌شه. برای همین، تابع set\_banners هم صدا زده می‌شه تا وضعیت پرچم‌ها رو آپدیت کنه.
* در نهایت، وضعیت جدید بازی شامل کارت‌ها و اطلاعات بازیکن‌ها (مثل کارت‌های جمع‌شده و پرچم‌ها) رو برمی‌گردونه.
  + نتیجه رو به صورت وضعیت شبیه‌سازی‌شده برمی‌گردونه.

****

**الگوریتم مینیمکس با هرس آلفا-بتا (تابع minimax)**

این تابع از یه الگوریتم جستجو استفاده می‌کنه تا بهترین حرکت رو پیدا کنه:

1. **پایه‌ی بازگشتی:**

اول از همه، بررسی می‌کنه که عمق جستجو تموم شده یا حرکت دیگه‌ای باقی نمونده. اگه تموم شده باشه، وضعیت فعلی بازی رو با تابع evaluate\_state امتیازدهی می‌کنه و برمی‌گردونه.

2. **اگه نوبت بازیکن ما (Maximizing) باشه:**

• تابع همه حرکت‌های ممکن رو بررسی می‌کنه.

• هر حرکت رو با تابع simulate\_move شبیه‌سازی می‌کنه تا وضعیت جدید بازی رو به‌دست بیاره.

• بعد، به‌صورت بازگشتی Minimax رو برای حریف (Minimizing) صدا می‌زنه.

• بهترین امتیاز رو نگه می‌داره و بقیه رو دور می‌ریزه.

• از هرس آلفا-بتا استفاده می‌کنه تا حرکت‌هایی که بررسی‌شون بی‌فایده‌ست رو رد کنه و سرعتش بره بالا.

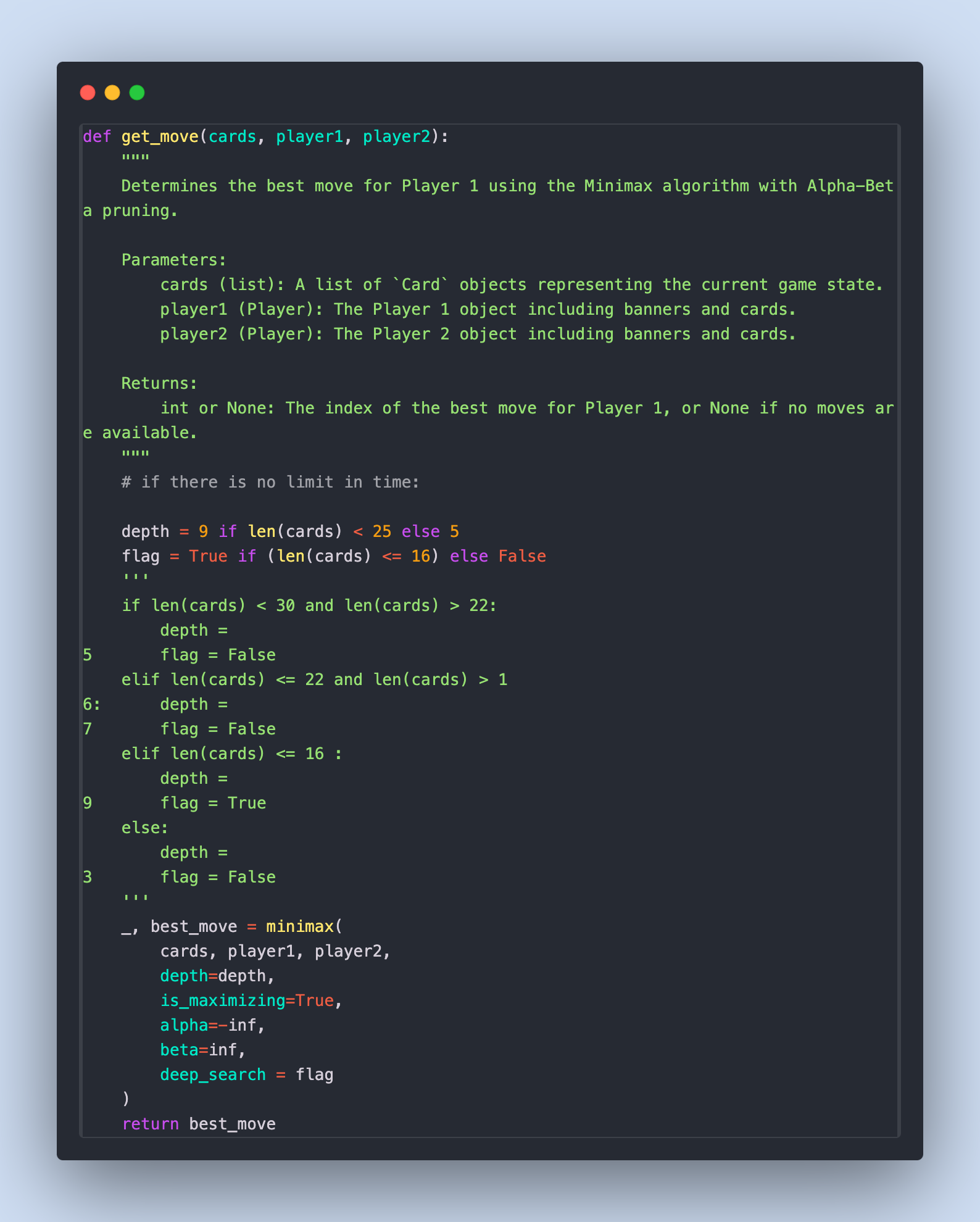
3. **اگه نوبت حریف (Minimizing) باشه:**

• دقیقاً مثل مرحله بالا عمل می‌کنه، ولی هدفش اینه که بدترین امتیاز ممکن رو برای بازیکن ما انتخاب کنه.

• باز هم با هرس آلفا-بتا، حرکت‌های غیرضروری رو رد می‌کنه.

4. **برگشت:**

• بعد از بررسی همه حرکت‌ها، تابع بهترین امتیاز و حرکت مربوط به اون امتیاز رو برمی‌گردونه.

****

این تابع آخرین مرحله‌ایه که ایجنت تصمیم می‌گیره “الان باید چه حرکتی کنم؟”. با کمک الگوریتم Minimax و تنظیمات خاصی که براساس وضعیت فعلی بازی انانجام می‌شه، بهترین حرکت ممکن رو برای بازیکن اول (ایجنت هوشمند ما) پیدا می‌کنه.

1. **بررسی تعداد کارت‌های باقی‌مونده:**

• تابع اول نگاه می‌کنه چندتا کارت روی تخته باقی مونده. تعداد کارت‌ها تعیین می‌کنه که عمق جستجوی Minimax چقدر باشه:

• **کارت‌های زیاد:** عمق کمتر (مثلاً 5) چون بررسی همه‌ی حالت‌ها زمان‌بر می‌شه.

• **کارت‌های کم:** عمق بیشتر (مثلاً 9) چون بازی داره به انتها نزدیک می‌شه و بررسی دقیق‌تر لازمه.

2. **فعال‌سازی حالت Deep Search:**

• وقتی تعداد کارت‌ها خیلی کم می‌شه (مثلاً کمتر از 16 کارت)، یه حالت خاص به اسم “Deep Search” فعال می‌شه. این حالت به جای اینکه بخوایم از تابع هیوریستیک استفاده کنیم مستقیما از خود استیت نهایی بازی استفاده میکنیم برای اینکه مشخص کنیم کی بازیو برده یا باخته.

3. **اجرای Minimax:**

• بعد از تنظیم عمق و حالت جستجو، تابع minimax رو صدا می‌زنه و ازش می‌خواد که بهترین حرکت رو بر اساس وضعیت فعلی بازی پیدا کنه.

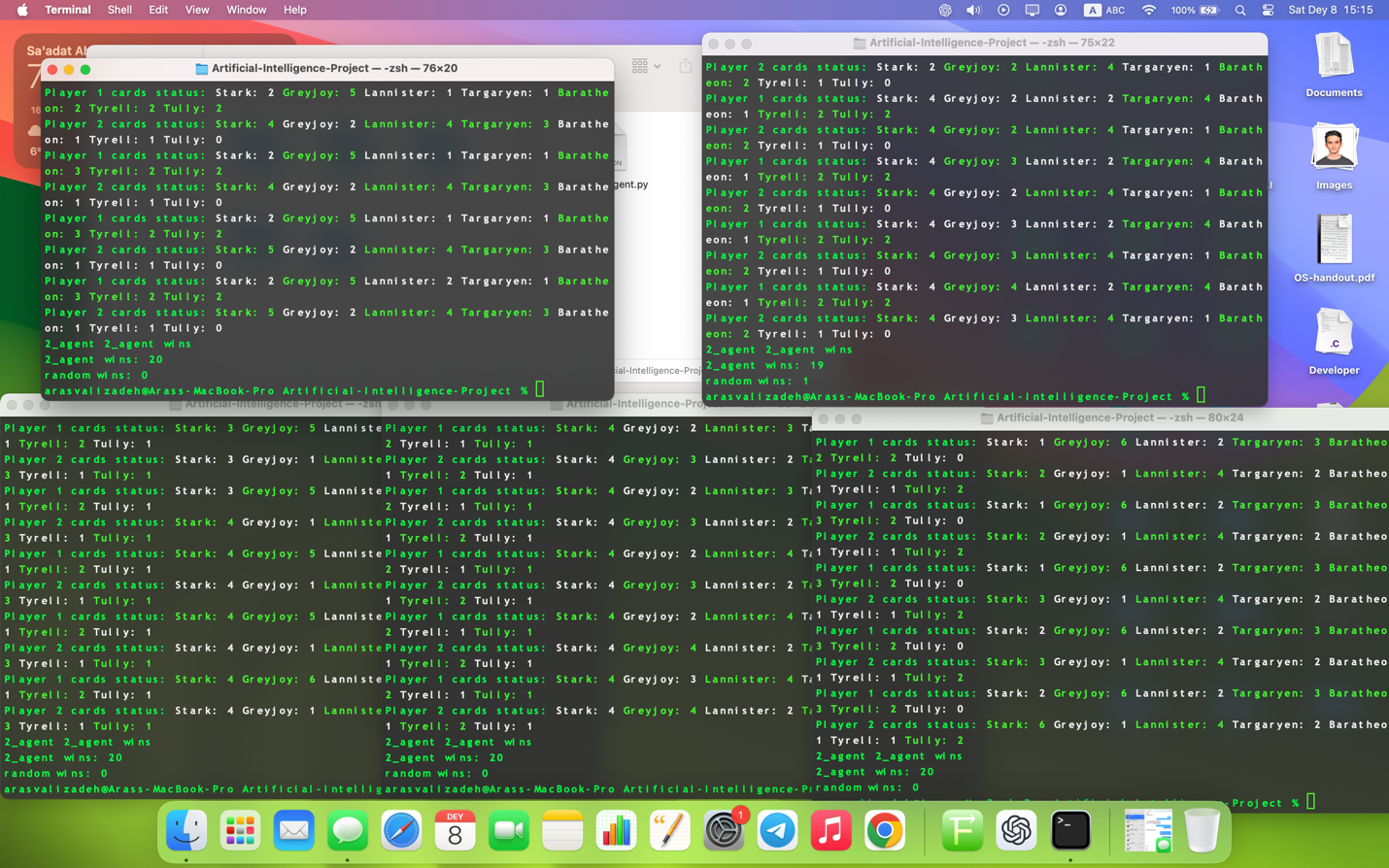
• Minimax برمی‌گردونه که کدوم حرکت بهترین امتیاز رو داره.

4. **برگشت بهترین حرکت:**

• وقتی Minimax نتیجه رو داد، این تابع شماره‌ی بهترین حرکت رو برمی‌گردونه تا ایجنت حرکتش رو انانجام بده.

**نتایج**:  
  
در نهایت ۱۰۰ بار ایجنت خودمون رو با ایجنت رندوم بازی دادیم که موفق شد ۹۹ بار ایجنت رندوم رو شکست بده و فقط ۱ بار ببازه. هرچی عمق مینیماکس رو بیشتر کنیم طبعا درصد موفقیت بیشتر میشه ولی از اونور زمان اجرای بازی هم طولانی میشه.

اسکرین شات نتایج خروجی رو پایین قرار میدم.



ممنون از توجه شما :‌)