دانشگاه صنعتی امیرکبیر
(پلی‌تکنیک تهران)
دانشکدهٔ ریاضی و علوم‌کامپیوتر

گزارش ۴: پیاده‌سازی یک بازی با روش‌های جستجوی تخاصمی

نگارش
نیما حسینی دشت بیاض

استاد
دکتر مهدی قطعی

فروردین ۱۴۰۰

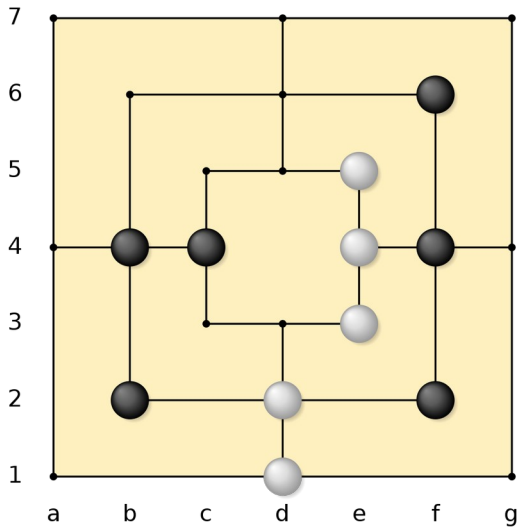
مقدمه

الگوریتم‌های جستجوی تخاصمی شرایطی را شبیه‌سازی می‌کنند که دو عامل سعی می‌کنند با یک‌دیگر مقابله کنند و جلوی سود دیگری را بگیرند. چنین شرایطی را می‌توان در بسیاری از بازی‌ها مشاهده کرد. در بازی‌ای مانند دوز، هر فرد با علم به این که طرف مقابل می‌خواهد جلوی پیشرفت او را بگیرد، سعی می‌کند تصمیمی بگیرد که بیشترین سود را برای خودش داشته باشد و همچنین جلوی پیشرفت رقیب را بگیرد. در نتیجه، بازی‌ها مثال‌های خوبی برای آشنایی با شرایط تخاصمی و این دسته از الگوریتم‌ها هستند.

در این تمرین به پیاده‌سازی بازی Nine Men's Morris[1] با استفاده از روش‌های تخاصمی و الگوریتم Alpha-Beta Pruning به همراه هیوریستیک می‌پردازیم.

بازی Nine Men's Morris

در این بازی یک تخته مانند شکل ۱ وجود دارد و هر بازیکن باید ۹ مهره‌ی خود را روی آن قرار دهد. مهره‌های یک بازیکن سیاه و دیگری سفید است. هر بار که یک بازیکن بتواند ۳ مهره‌ی خود را به طور افقی یا عمودی در یک خط قرار دهد، می‌تواند یکی از مهره‌های حریف را از بازی خارج کند. به این حالت یک Mill می‌گویند. البته مهره‌ای که از حریف برای خارج کردن انتخاب می‌کند، خودش نباید یک Mill تشکیل داده باشد؛ مگر این‌که انتخاب دیگری وجود نداشته باشد. هرگاه یک بازیکن فقط ۲ مهره‌اش در زمین بماند یا همه‌ی مهره‌هایش به‌گونه ای محاصره شده باشند که نتواند آن‌ها را جابه‌جا کند، بازی تمام می‌شود و حریف او برنده است.



عکس ۱: تخته‌ی بازی Nine Men's Morris

[1] https://en.wikipedia.org/wiki/Nine_men%27s_morris

بازی از سه مرحله یا فاز تشکیل شده است.

۱. در مرحله‌ی اول، بازیکن‌ها به نوبت مهره‌های خود را روی تخته می‌چینند. اگر ۳ مهره کنار هم قرار بگیرد، Mill تشکیل می‌دهد و یک مهره‌ی حریف حذف می‌شود.

۲. پس از ورود همه‌ی مهره‌ها، بازی با جابه‌جا کردن مهره‌ها ادامه می‌یابد. مهره‌ها فقط می‌توانند به خانه‌های همسایه‌ی خود منتقل شوند و امکان پریدن از روی مهره‌های دیگر وجود ندارد. در هر خانه هم فقط یک مهره قرار می‌گیرد. در این مرحله هم، مشابه مرحله‌ی قبل، می‌توان Mill تشکیل داد.

۳. در صورتی که مهره‌های یک بازیکن به سه عدد برسد، وارد فاز سوم می‌شویم و آن بازیکن می‌تواند مهره‌ی خود را به هر مکان خالی‌ای (و نه فقط خانه‌های همسایه) روی تخته جابه‌جا کند. به این کار پرواز یا fly می‌گویند. در این مرحله هم امکان تشکیل Mill وجود دارد.

در نهایت بازیکنی که ۲ مهره باریش باقی بماند بازنده است.[1]

پیاده‌سازی بازی

در ادامه جزئیات پیاده‌سازی بازی با استفاده از الگوریتم Alpha-Beta Pruning در زبان پایتون را بررسی می‌کنیم. کد کامل این برنامه در آدرس گیت‌هاب زیر قرار دارد.

https://github.com/nimahsn/AI-Course-Projects/tree/main/report04-AdversarialSearch

ابتدا پیاده سازی الگوریتم را بررسی و سپس جزئیات مدل مسئله را بررسی می‌کنیم.

پیاده‌سازی الگوریتم

در این برنامه، از الگوریتم Alpha-Beta Pruning برای انجام جستجو در درخت بازی استفاده شده است. این الگوریتم نسخه‌ای از الگوریتم Minimax است که در آن برخی شاخه‌ها و از درخت حذف و بررسی نمی‌شوند. همچنین برای بهبود زمان اجرا و مصرف حافظه، از روش‌های Evaluation و Cut off نیز استفاده شده است. در این روش‌ها، به جای پیمایش درخت بازی تا آخرین مرحله[۲] و محاسبه‌ی سود یا زیان نهایی، از روشی هیوریستیک استفاده می‌شود تا میزان سود در هر مرحله تخمین زده شود. همچنین روش Cut off مشخص می‌کند که تا چه عمقی از درخت پایین برویم و سپس پیش‌بینی را انجام دهیم.[2]

الگوریتم در کلاس AlphaBetaPlayer و در فایل ai_player.py پیاده سازی شده است. این کلاس نشان‌دهنده‌ی یک بازیکن مجازی است و رنگ مهره‌هایش در بازی در فیلد black مشخص می‌شود. (به طور

۲ Terminal State

True یا False.) الگوریتم از سه تابع اصلی alpha_beta_search, max_value و min_value تشکیل می‌شود[۳]. همچنین تابع cut_off_test نیز در همین کلاس قرار دارد. تابع alpha_beta_search با دریاف وضعیت فعلی بازی، کار تصمیم گیری را شروع می‌کند. هر کدام از توابع max_value و min_value نشان‌دهندی عملیات max و min در الگوریتم Minimax هستند. هر کدام از این دو به ترتیب بیشترین و کمترین امتیاز از گره‌های عمق پایین تر درخت به همراه گره متناظر را باز می‌گردانند. در ابتدای آن‌ها توابع cut_off_test و evaluate صدا زده می‌شود. تابع cut_off_test با استفاده فیلد depth در کلاس و یا تشخیص terminal بودن گره، تصمیم می‌گیرد که عملیات evaluation انجام شود یا خیر. تابع evaluate نیز در کلاس Evaluator پیاده سازی شده است و با محاسبه‌ی چندین تابع هیوریستیک دیگر، سود را پیش‌بینی می‌کند.

```python
class AlphaBetaPlayer:
    def __init__(self, black: bool, evaluator: "Evaluator", cut_off_depth: int = 3, opponent: "AlphaBetaPlayer" = None)

    def alpha_beta_search(self, state: "Board") -> "Board":
        score, new_state = self.max_value(state, -math.inf, math.inf, 1)
        return score, new_state

    def max_value(self, state: "Board", alpha: int, beta: int, depth: int) -> Tuple[int, "Board"]:
        if self.cut_off_test(state, depth):
            return (self.evaluator.evaluate(state, self.black), state)
        v = (-math.inf, None)
        all_s = list(state.get_all_moves(self.black))
        shuffle(all_s)
        for new_state in all_s:
            (score_minval, _) = self.min_value(new_state, alpha, beta, depth+1)
            max_value = max(v[0], score_minval)
            if max_value == v[0]:
                pass
            elif max_value == score_minval:
                v = (score_minval, new_state)
            if v[0] >= beta:
                return v
            alpha = max(alpha, v[0])
        return v
```

عکس ۲: قسمتی از پیاده‌سازی کلاس AlphaBetaPlayer و الگوریتم آن.

هیوریستیک‌های الگوریتم

همان‌طور که در بالا گفته شد، برای پیش‌بینی سود نیاز به یک سری از توابع هیوریستیک داریم. این توابع در کلاس Evaluator قرار گرفته‌اند و تابع evaluate مجموع آن‌ها را حساب می‌کند. این هیوریستیک‌ها عبارتند به شرح زیر می‌باشند.

۱. eval_diff_markers_in: این تابع، تفاضل تعداد مهره‌های داخل تخته‌ی بازیکن و حریف را محاسبه می‌کند[۴].

۲. eval_last_mill: این تابع در صورتی که در آخرین حرکت، یک Mill برای بازیکن ایجاد کرده باشد

۳ پیاده سازی الگوریتم براساس شبهه کد موجود در کتاب Artificial Intelligence : a Modern Approach است.

۴ در کد از کلمه‌ی marker برای شاره به مهره‌ها استفاده شده است.

مقدار ۱، اگر حریف یک Mill به دست آورده باشد مقدار -۱ و در غیر این صورت ۰ را باز می‌گرداند.

۳. eval_diff_blocked_markers: این تابع تفاضل تعداد مهره‌های بلوکه شده‌ی بازیکن و حریف را حساب می‌کند. این‌ها مهره‌هایی هستند که محاصره شده‌اند و نمی‌توانند حرکت کنند.

۴. eval_diff_double: این تابع تفاضل تعداد چینش‌های دوتایی بازیکن و حریفش را حساب می‌کند. چینش‌های دوتایی، چینش‌هایی هستند که دو مهره‌ی بازیکن طوری کنار هم قرار گیرند که با یک مهره‌ی دیگر تشکیل Mill بدهند.

۵. eval_diff_mill_count: این تابع تفاضل کل تعداد Mill های بازیکن و حریفش را محاسبه می‌کند.

۶. eval_win: در نهایت این تابع در صورتی که بازیکن در این مرحله بازی را برده باشد، مقدار ۱، اگر باخته باشد، -۱ و اگر بازی ادامه یابد صفر را برمی‌گرداند.

تابع evaluate با توجه به فاز بازی، تعدادی از این توابع را انتخاب کرده و ترکیبی خطی از آن‌ها را به عنوان سود یک state حساب می‌کند. به طور مثال در فاز سوم بازی، از آن‌جایی که مهره‌های حریف می‌توانند پرواز کنند، block کردن مهره‌ها ارزشی ندارد، پس در این فاز از این هیوریستیک استفاده نمی‌شود. از طرفی در فاز اول بازی، چون تعداد مهره‌ها در حال افزایش است، شانس بلاک کردن مهره‌های حریف بیشتر است، پس این هیوریستیک استفاده شده است. در عکس زیر ضرایب هر هیوریستیک در هر فاز مشخص است[۵] [3].[۶]

```
61  class Evaluator:
62      def __init__(self) -> None:…
64
65      def _eval_diff_markers_in(self, board: "Board", black: bool) -> int:…
71
72      def _eval_last_mill(self, board: "Board", black: bool) -> int:…
81
82      def _eval_diff_blocked_markers(self, board: "Board", black: bool) -> int:…
98
99      def _eval_win(self, board: "Board", black: bool) -> int:…
119
120     def _eval_diff_double(self, board: "Board", black: bool):…
140
141     def _eval_diff_mill_count(self, board: "Board", black: bool):…
146
147     def evaluate(self, board: "Board", black: bool):
148         if board.phase == 1:
149             return 20*self._eval_last_mill(board, black) +\
150                 25*self._eval_diff_mill_count(board, black) +\
151                     2*self._eval_diff_blocked_markers(board, black) +\
152                         5*self._eval_diff_markers_in(board, black) +\
153                             8*self._eval_diff_double(board, black)
154
155         elif board.phase == 2:
156             return 15*self._eval_last_mill(board, black) +\
157                 35*self._eval_diff_mill_count(board, black) +\
158                     10*self._eval_diff_blocked_markers(board, black) +\
159                         15*self._eval_diff_markers_in(board, black) +\
160                             2000*self._eval_win(board, black)
161
162         elif board.phase == 3:
163             return 3000*self._eval_win(board, black) +\
164                 18*self._eval_last_mill(board, black)
```

عکس ۳: خلاصه‌ای از پیاده‌سازی هیوریستیک‌ها و کلاس Evaluator

۵ ممکن است این ضرایب در پروژه‌ی روی گیت‌هاب با عکسی که هنگام نگارش این گزارش گرفته شده است، متفاوت باشد.

۶ این ضرایب و هیوریستیک‌ها با اقتباس از منبع ذکر شده استفاده شده‌اند، اما مقدار آن‌ها پس از چندین بار امتحان عوض شده است.

پیاده‌سازی مدل مسئله

مدل بازی در کلاس Board و در فایل board.py پیاده شده است. در این برنامه، تخته را با ۷ سطر و ۷ ستون نمایش می‌دهیم که موقعیت‌های مجاز آن در لیست positions قرار دارد. همسایه‌های هر موقعیت در دیکشنری adjacent_dict و چینش‌های که به ازای هر موقعیت یک Mill تشکیل می‌دهند در دیکشنری mill_dict آورده شده است.[4] کلاس Board برای ذخیره‌ی موقعیت مهره‌ها از یک دیشکنری استفاده می‌کند که کلیدهای آن موقعیت‌های مجاز تخته و مقادیر آن آبجکت‌های کلاس Marker هستند. در صورتی که مهره‌ای در یک موقعیت وجود نداشته باشد، مقدار None در دیکشنری ذخیره می‌شود. رنگ هر مهره هم در کلاس Marker مشخص می‌شود. این کلاس مجموعه‌ای از فیلدهای دیگر را نیز دارد که برای محاسبه‌ی هیوریستیک‌ها و تشخیص فاز بازی کاربرد دارند. چهار تابع اصلی در این کلاس قرار دارد که وظیفه‌ی پیدا کردن تمام action های مجاز برای وضعیت فعلی تخته را دارند و با اعمال این action ها، کپی‌های جدیدی از کلاس Board را برمی‌گردانند. سه تا از این توابع برای هر یک از سه فاز اصلی بازی طراحی شده‌اند و یک تابع هم وظیفه دارد با توجه به فاز بازی، تابع مناسب را فراخوانی کند. دو تابع کمکی is_mill و remove_opp_marker هم به ترتیب برای تشخیص Mill و محاسبه‌ی تمام حالات ممکن برای حذف مهره‌ی حریف نوشته شده‌اند. تمامی این توابع وظیفه دارند که علاوه‌بر ساخت حالات جدید، فیلدهای کلاس را هم به‌روز کنند. تابع __repr__ وظیفه‌ی ایجاد یک نمایش یا representation برای تخته را دارد.[5]

```
78  class Board:
79      def __init__(self, phase: int = 1, dict_markers = None) -> None:
80          self.pos_marker_dict: Dict[Tuple[int, int], "Marker"] = dict_markers
81          if self.pos_marker_dict == None:
82              self.pos_marker_dict = {}
83              for pos in positions:
84                  self.pos_marker_dict[pos] = None
85          self.phase = phase
86          self.black_markers_out = 9
87          self.white_marker_out = 9
88          self.black_markers_in = 0
89          self.white_markers_in = 0
90          self.white_markers_captured = 0
91          self.black_markers_captured = 0
92          self.white_mills = 0
93          self.black_mills = 0
94          self.last_move_is_black: bool = None
95          self.last_move_type: int = None
96
97      def get_all_moves(self, black):…
107
108     def get_all_moves_phase1(self, black):…
137
138     def get_all_moves_phase2(self, black):…
170
171     def get_all_moves_phase3(self, black):…
210
211     def is_mill(self, position, black) -> bool:…
219
220     def remove_opp_marker(self, black) -> "Board":…
257
258     def is_terminal(self):…
266
267     def __repr__(self) -> str:…
```

عکس ۴: چکیده‌ی کلاس Board و فیلدهای آن

استفاده از الگوریتم‌ها

دو تابع ai_vs_ai و ai_vs_human در فایل main.py نوشته شده است که با آن‌ها می‌توان بازی را اجرا کرد. همانطور که از اسم توابع مشخص است، در اولی هر دو بازیکن از هوش مصنوعی استفاده می‌کنند و بازیکن‌ها از کلاس AlphaBetaPlayer هستند. در تابع دوم، یک بازیکن از هوش مصنوعی و دیگری کاربر کامپیوتر است که با استفاده از محیط ترمینال با کامپیوتر بازی می‌کند. برای نشان دادن کاربر بازیکن انسان، کلاس HumanPlayer طراحی شده است که توابعی را برای گرفتن ورودی از کاربر و اعمال حرکت‌ها برروی تخته پیاده می‌کند. این توابع مستقل از توابع get_move که در کلاس Board پیاده شده بودند، عمل می‌کنند و مانند آن‌ها باید فیلدهای کلاس Board را به‌روز کنند. تابع prompt_and_move در این کلاس با گرفتن وضعیت تخته، تابع مناسب را با توجه به فاز بازی فراخوانی می‌کند و با گرفتن ورودی از کاربر، در صورت معتبر بودن حرکت، یک کپی جدید از تخته ایجاد شده و حرکت روی آن اعمال می‌شود.

```
You, a day ago | 1 author (You)
4   class HumanPlayer:
5   >   def __init__(self, black) -> None:…
7
8   >   def _get_action_input(self, phase):…
29
30  >   def _get_mill_input(self):…
36
37  >   def _prompt_and_move_phase1(self, board: "b.Board"):…
76
77  >   def _prompt_and_move_phase2(self, board: "b.Board"):…
119
120 >   def _prompt_and_move_phase3(self, board: "b.Board"):…
164
165 >   def prompt_and_move(self, board: "b.Board") -> "b.Board":…
```

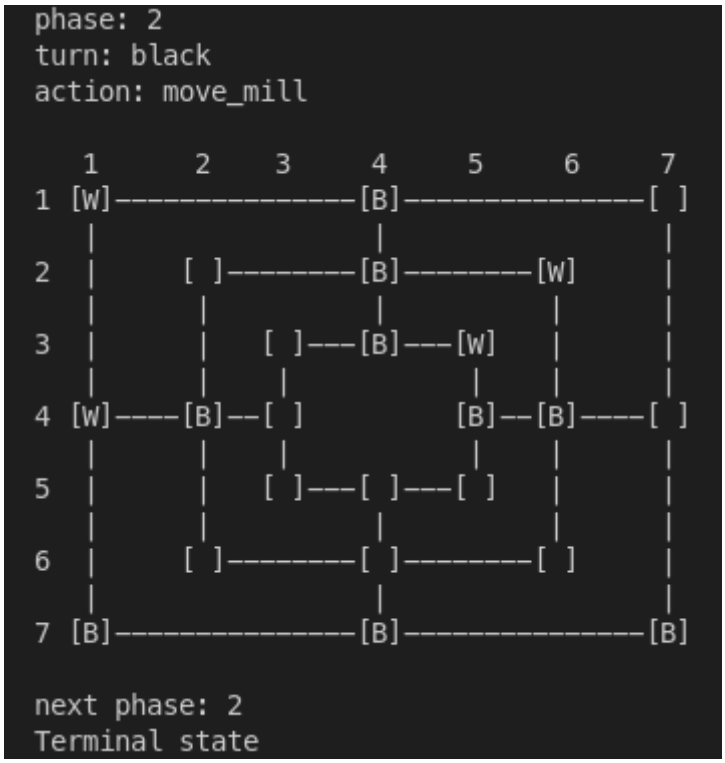
عکس ۵: چکیده‌ی کلاس HumanPlayer

در نهایت با استفاده از این کلاس و کلاس AlphaBetaPlayer، دو تابع ai_vs_ai و ai_vs_human بازی را اجرا می‌کنند. هر بار حرکت بازیکن AI با فراخوانی تابع alpha_beta_search و هر حرکت بازیکن انسان، با تابع prompt_and_move انجام می‌شود. بعد از هر حرکت بازیکن‌ها، تخته چاپ شده و اطلاعاتی مانند فاز، آخرین حرکت و... نمایش داده می‌شود. همچنین عمق Cut off الگوریتم به عنوان آرگومان این دو تابع دریافت می‌شود.

نمونه‌ی اجرا

در ادامه نمونه‌ای از اجرای بازی در حالت AI Vs AI با عمق توقف ۳ آورده شده است. به دلیل طولانی بودن

بازی، تنها وضعیت نهایی تخته نمایش داده شده است. در این بازی، بازیکن با مهره‌های سیاه (B)، تمام مهره‌های سفید (W) را block کرده و برنده‌ی بازی است.



عکس ۶: حالت نهایی یک بار اجرای بازی توسط دو بازیکن AI

در حالت Human Vs AI، هر بار باید حرکت مورد نظر در ترمینال وارد شود. در فاز اول، دو عدد با یک فاصله وارد می‌شود و مهره‌ی بازیکن در آن مختصات قرار می‌گیرد (نمونه ورودی: 5 3 یعنی سطر ۳ و ستون ۵). در فاز دوم، در دو مرحله، دومختصات مبدا و مقصد حرکت را باید وارد کرد (مانند نمونه ورودی بالا). ورودی فاز سوم نیز مانند فاز دوم باید وارد شود. در هر کدام از این فازها، در صورت ایجاد یک mill، مختصات مهره‌ی حریف که باید حذف شود نیز از کاربر پرسیده می‌شود که باید مثل قبل وارد شود.

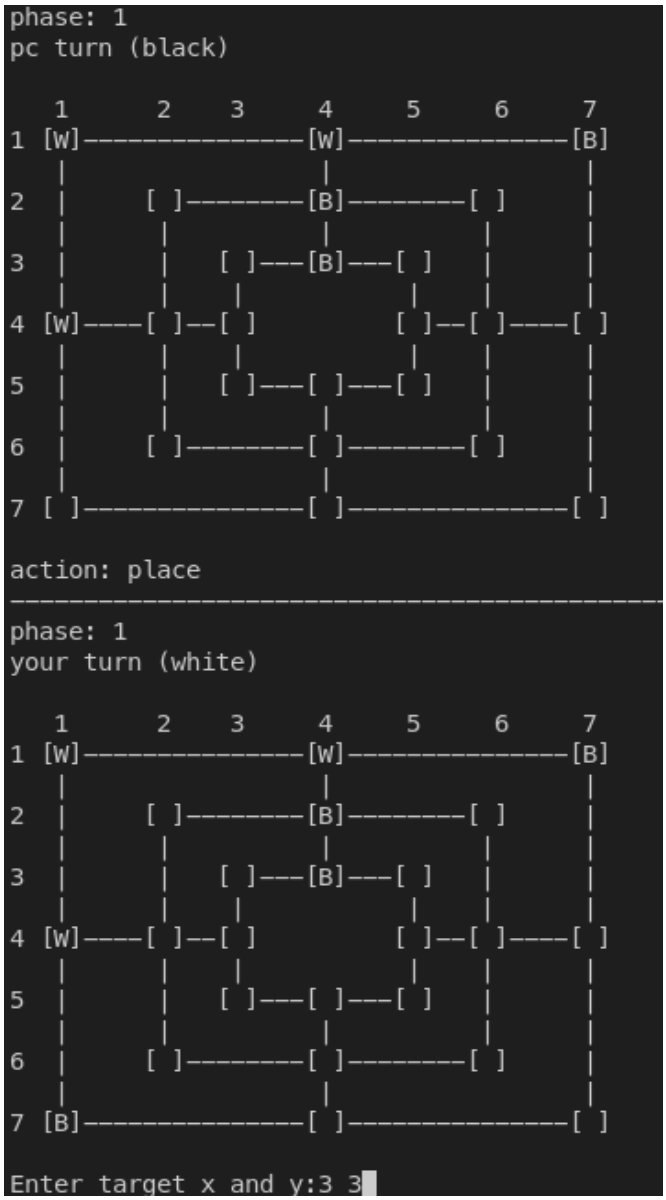
عکس ۷ یک مرحله‌ی میانی در فاز اول از اجرای بازی در حالت Human Vs AI را نشان می‌دهد. همانطور که مشاهده می‌شود، بازیکن سیاه (AI) سعی می‌کند جلوی بازیکن سفید (انسان) برای تشکیل Mill را بگیرد.

جمع‌بندی

در این پروژه بازی Nine Men’s Morris را با استفاده از الگوریتم‌های تخاصمی و روش‌های هیوریستیک پیاده سازی کردیم. در چندین اجرای این بازی در حالت AI vs AI، از آن‌جایی‌که هر دو بازیکن از یک الگوریتم استفاده می‌کنند و هر دو تصمیم بهینه را می‌گیرند، بازی در بسیاری از مواقع بسیار به طول می‌انجامید. همچنین با رسیدن به فاز سوم بازی و به‌دست آوردن قابلیت پرواز، بازیکن‌ها سعی می‌کردند با استفاده از

این قابلیت، جلوی تشکیل Mill توسط حریف را بگیرند و این خود باعث طولانی‌تر شدن بازی می‌شد.

در حالت Human Vs AI نیز گرچه عملکرد کلی الگوریتم در مقابل انسان ضعیف‌تر است (با عمق برش ۴)؛ اما با رسیدن به فاز دو و سه، الگوریتم به خوبی جلوی تشکیل Mill توسط انسان را می‌گرفت که باز هم منجر به طولانی‌تر شدن بازی می‌شد.



عکس ۷: نمونه‌ی اجرای بازی در حالت Human Vs AI

منابع

۱ https://www.thesprucecrafts.com/nine-mens-morris-board-game-rules-412542

۲ Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice .Hall, 2010

۳ https://kartikkukreja.wordpress.com/2014/03/17/heuristicevaluation-function-for-nine-mens-morris/

۴ https://github.com/JakobDomislovic/Nine_Mens_Morris_with_Franka_ROS

۵ https://github.com/rajko-z/nine-mans-morris-python