به‌نام خدا

گزارش پروژه‌ی نهایی

درس «مقدمه‌ای بر هوش مصنوعی»

نرّاد

(بازی‌کننده‌ی تخته‌نرد[1])

با استفاده از الگوریتم‌های

Expecti MiniMax، α-β و Genetic Algorithm

سارا احمدیان (۸۳۱۱۷۸۶۵)

آیدین نصیری‌شرق (۸۳۱۱۳۸۰۴)

بهمن و اسفند ۱۳۸۵

[1] BackGammon

نمایه

۱ توضیحات عمومی

۱/۱ معرفی بازی

بازی تخته‌نرد، یک بازی دو نفره‌ست که در آن هر شخص ۱۵ مهره روی صفحه داشته و سعی می‌کند با انداختن تاس و اتّخاذ تدبیر مناسب، زودتر از حریف مقابل مهره‌هایش را به خارج از صفحه منتقل کند. قوانین دقیق بازی در فایل `backgammon.doc` که در سایت درس موجودست ذکر شده است.

هم‌چنین در آن فایل، نحوه‌ی گرفتن ورودی و خروجی (خواندن تاس‌ها و حرکات حریف از ورودی، نوشتن حرکات خودِ بازی‌کن در خروجی) به تفصیل آمده است.

۱/۲ تغییرات اِعمال شده نسبت به قوانین صحبت با قاضی[۲]

با توجه به جزئیات تعیین شده در فایل قوانین، برخی ابهامات نظیر نحوه‌ی اطلاع‌رسانی به بازی‌کن از شروع بازی و نیز نحوه‌ی بیان تمام‌شدن بازی به قاضی به‌صورت زیر تثبیت گردید. فلذا، مطلوبست تغییرات حاصله در قاضی نهایی نیز اِعمال گردند.

- **شروع بازی:** در اوّلین حرکت بازی، به بازی‌کن شروع‌کننده، تنها دو تاس داده می‌شود و وی این مهم را به‌فال آغازه می‌گیرد. واضح‌ست که در سایر حرکات، همواره حداقل دو تاس (کنونی و پیشین) به هر بازی‌کن ارسال می‌شود.
- **پایان بازی:** در انتهای بازی، پس از انجام آخرین حرکت که مهره‌های یکی از بازی‌کنان تمام شود، بازی‌کن باید رشته‌ی `finish` و سپس امتیاز خودش و امتیاز بازی‌کن دیگر را بنویسد. امتیاز بازنده همواره برابر صفر و امتیاز برنده، طبق قوانین گفته شده برابر یک یا دو[۳] می‌باشد (حالت امتیاز ۳[۴] به‌دلیل تأثیر زیاد در تشدید نخبه‌پروری در «الگوریتم ژنتیک» درنظر گرفته نشده‌ست).
- **ترتیب خواندن و نوشتن حرکات:** از آن‌جا که ممکن‌ست ترتیب حرکات الزاماً ترتیب تاس‌ها نباشد این ترتیب چه برای خواندن و چه برای نوشتن، دل‌خواه درنظر گرفته شده است. نیز به هنگام بررسی[۵] حرکات انجام شده توسّط حریف (در ورودی) تمامی جای‌گشت‌های ممکن درنظر گرفته می‌شود.
- **توضیحات اضافه:** در صورت دریافت کردن یک حرکت اشتباه از سوی قاضی (که از حریف گرفته‌ست) و نیز در خاتمه‌ی بازی، بازی‌کن ممکن‌ست توضیحات اضافه در جریان خطای استاندارد[۶] بنویسد که با اجرای برنامه به‌صورت `bg.exe 2>/dev/null` این مفصّلات نمایش داده نمی‌شوند.

۱/۳ انتخاب زبان برنامه‌سازی (C++)

برای پیاده‌سازی این بازی از زبان C++ به‌دلیل قدرت بالا در پیاده‌سازی مختصر و جامع یک برنامه‌ی شی‌گرا[۷] (که رویه‌ی مدّنظر در نگارش برنامه بوده‌است) استفاده شده‌است. شرح مختصری از کلاس‌ها، توابع و جزئیات بازی

---

[2] Judge
[3] gammon
[4] backgammon
[5] Verficiation
[6] Standard Error Stream

در فصل دوم آمده‌است. به‌دلیل یک‌سان سازی سایر بخش‌ها (نظیر الگوریتم ژنتیک و ...) و نیز استفاده از بازی‌انداز caiaio (که در فصل چهارم شرح داده شده‌است)، در آن‌ها نیز از زبان c++ استفاده شد. متأسفانه تنها مشکل این زبان، عدم امکان اتّصال مستقیم با استفاده از ساکت‌ها بود که این عمل به‌طور موقّت با استفاده از caiaio میسّر گشت.

۱/۴ محیط کار و آزمایش

نتایج به‌دست‌آمده در این پروژه، تماماً بر روی یک سیستم Intel PIII 800MHz در سیستم‌عامل Windows XP، با حافظه‌ی RAM برابر ۲۵۶ مگابایت بوده است.

۱/۵ اجرای برنامه

این برنامه در ویرایش‌گر[8] Emacs نوشته‌شده و توسّط کامپایلر g++ 3.4.2 در بسته‌ی MinGW 3.2.0 در سیستم‌عامل Windows آماده‌سازی، تست و اجرا شده‌است.

برای کامپایل و اجرای برنامه در محیط Windows، کافی‌ست فایل دسته‌ای[9] `gg.bat` اجرا شود که در آن محل قرارگیری کامپایلر به‌طور پیش‌فرض «`c:\mingw\bin\g++`» تنظیم شده‌ست و در صورت نیاز لازم‌ست تغییر یابد.

برای انجام بازی مابین دو نسخه از برنامه، کافی‌ست فایل `ca.exe` اجرا شود. نام برنامه‌ها از فایل `manager.txt` خوانده می‌شود. توضیحات بیش‌تر در بخش «انجام بازی‌ها» آمده‌ست.

نهایتاً کامپایل و اجرای الگوریتم ژنتیک توسّط فایل دسته‌ای `ga.bat` انجام می‌شود که نیازمندی‌های آن مشابه `gg.bat` است.

---

[7] Object Oriented
[8] Editor
[9] batch file

۲ جزئیات پیاده‌سازی بازی (فایل bg . cpp)

۲٫۱ کلاس‌های برنامه

به‌دلیل رعایت رهیافت شی‌گرایی (Object Orientation)، کلاس‌های زیر برای اشیاء تعریف شدند:

۲٫۱٫۱ تعریفِ‌نوع[10] Error

`Error` در اصل یک تعریفِ‌جدید و ساده از نوع[11] `ostringstream` است که به‌طور مشخّص در توابع `CheckOneMove` و `CheckInst` کاربرد دارد. در اشیاء ساخته‌شده از این (به‌اصطلاح) کلاس می‌توان مانند جریان‌های رایج `cerr` و `cout`، اشیاء را با عمل‌گر >> ریخت و سپس مقادیر ریخته‌شده در آن شیء را، بعداً توسّط تابع `str()`. به‌صورت `char *` خواند یا نوشت. مهم‌ترین تسهیلی که در «جریان» گرفتن این کلاس به‌جای یک نوع «رشته»ای به‌وجود می‌آید، امکان جای‌گزینی سریع آن با `cerr` و نیز امکان ارسال بی‌دغدغه‌ی اشیاء با انواع مختلف (نظیر `int` و سپس `string`) است.

۲٫۱٫۲ ترتیب PlayerSign

که شامل ۳ نام‌اصلی و شناسایی برای بازیکنان است. این نام‌ها `NOP` (مخفّف `No Player`، برای جلوگیری از انتساب خانه‌های خالی و اوّلیه به بازیکنان اصلی)، `X` و `O` است.

۲٫۱٫۳ کلاس Player

که شامل ۳ بازیکن اصلی `NOP`، `player[1]` یا `player[X]` و نهایتاً `player[2]` یا `player[O]` است. نیز از این کلاس ۲ اشاره‌گر `me*` و `he*` ساخته شده است که در طول بازی به‌ترتیب به بازیکنی که برنامه قرارست به‌جای آن بازی‌کند (`me`) و حریف وی (`he`) اشاره دارند. بسته به شروع‌کننده‌ی بازی، `me` به‌یکی از `player[X]` یا `player[O]` اشاره دارد.

متغیّرهای درونی این کلاس، تنها در برگیرنده‌ی PlayerSign بازی‌کن هستند، حال آن‌که در getterهای این کلاس، بسته به `X` یا `O` بودن بازیکن، نام او (به‌صورت یک کاراکتر)، جهت حرکت در تخته (+۱ برای `X` و -۱ برای `O`)، خانه‌ی شروع (در صورت زده‌شدن، ۰ برای `X` و ۲۳ برای `O`) و نهایتاً اندیس درونی خانه‌ی بار و منزل برگردانده می‌شود. لازم به‌ذکرست که برای جلوگیری از تداخل و نیز سهولت تحلیل، برخلاف آن‌چه در سند مقاوله‌نامه[12] تنظیم‌شده، خانه‌های ۲۴ و ۲۵ (به‌جای بار و منزل هر دو بازیکن)، تنها بار و خانه‌ی `X` درنظر گرفته‌شدند و در دیگر سو، بار و منزل بازیکن `O` در سراسر برنامه برابر خانه‌های ۲۶ و ۲۷ انگاشته شده‌اند. این تغییر به‌هنگام تبادل اطّلاعات با قاضی در هنگام نوشتن و خواندن، به‌ترتیب در خطوط ۹۶ تا ۱۰۰ و نیز تابع `FixIndices()` در خطوط ۵۸۵ تا ۵۹۰ خنثی/ایجاد شده‌اند.

---

[10] `typedef`
[11] Type
[12] Protocol

۲٫۱٫۴ کلاس Move

این کلاس در برگیرنده‌ی یک حرکت از خانه‌ی FROM به‌خانه‌ی TO است. به‌دلیل سهولت تعامل (خواندن و نوشتن) با اشیاء این کلاس، عمل‌گرهای << و >> برای خواندن از یک istream و نوشتن در یک ostream تعریف‌مجدّد[۱۳] شده‌اند. نیز عمل‌گرهای > و == برای قراردادن اشیاء این کلاس در دربرگیرنده[۱۴]های مرتّب نظیر map و vector، تعریف شده‌اند.

۲٫۱٫۵ کلاس Instruction و InsPack

این دو کلاس فی‌الواقع وجود خارجی نداشته و یک تعریفِ‌نوع از یک بردار[۱۵] (از Move و Instruction) هستند. کلاس Instruction، یک آرایه‌ی پویا (داینامیک) از Moveها را در بر می‌گیرد. با این تعریف، تمام حرکات حریف در یک نوبت بازی و نیز حرکات تصمیم‌گرفته شده را می‌توان در یک شی از کلاس Instruction ریخت. از سوی دیگر، مجموعه‌ی حرکات ممکن برای یک وضعیت از بازی (با معلوم بودن تاس و بازی‌کنی که نوبت اوست) را می‌توان در یک InsPack ریخت؛ که این‌کار بر عهده‌ی تابع MaximalMoves() است.

برای این کلاس نیز عمل‌گرهای << و >> برای سهولت عمل با جریان[۱۶]های ورودی و خروجی تعریفِ‌مجدّد شده است.

۲٫۱٫۶ کلاس Dice

این کلاس به‌صورت یک آرایه‌ی پویا از اعداد (int) تعریف شده و محتوای یک تاس در یک نوبت را نگه‌داری می‌کند. برای کم‌کردن دغدغه‌های آتی که ممکن‌ست در کار با تاس‌های جفت پیش بیاید، در هنگام خواندن ورودی (که با تعریفِ‌مجدّد عمل‌گر << از جریان ورودی، حالت انتزاعی به‌خود گرفته)، چنین تاسی تبدیل به ۴ مقدار از عدد می‌شود (خطوط ۱۴۸ تا ۱۵۲).

۲٫۱٫۷ کلاس Column

دارای ۳ عضو می‌باشد که مبیّن تعداد، نوع (تعلّق به کدام بازی‌کن) و آفسِت تعیین‌شده (و بهینه، برای درنظرگرفتن بار و منزل هر بازیکن در شروع و پایان حرکات یک مهره روی صفحه) می‌باشند.

۲٫۱٫۸ کلاس Board

آخرین و اصلی‌ترین کلاس برنامه، کلاس Board است که تنها متغیر آن، یک آرایه‌ی ۲۸ عنصره از Column ها می‌باشد. از آن‌جا که برای تحلیل در اعماق، محاسبه، تحلیل صحّت و انجام حرکات بهینه، همواره به یک تخته (نه لزوماً تخته‌ی جاری) احتیاج داریم، توابع چنین کارهایی به‌صورت توابع عضو[۱۷] از این کلاس تعریف شده‌اند.

برای این کلاس، عمل‌گر >> برای نوشتن در خروجی (به‌صورت زیبا و نیمه‌گرافیکی)، تعریف شده‌است که نه تنها در اشکال‌زدایی‌کردن[۱۸] برنامه مورد استفاده واقع شده‌است، بلکه در حین بازی نیز تخته‌ی جاری را در جریان خطای استاندارد[۱۹] می‌نویسد.

---

[13] Overload
[14] Container
[15] vector
[16] Stream
[17] Member Function
[18] Debugging

۲٫۲ توابع عمومی برنامه

توابع ساده‌ی زیر، عمومی هستند.

۲٫۲٫۱ تابع void FixIndices(Instruction &)

در `Instruction` داده‌شده، اندیس خانه‌های بار و منزل بازی‌کن `O` را (در صورت وجود) به مقادیر مطلوب مقاوله‌نامه (برای ارسال به قاضی) تبدیل می‌کند. چنان‌که در بخش بعد گفته می‌شود، از آن‌جا که ارسال پارامتر با ارجاع است، این تغییرات روی همان عنصر اِعمال می‌شوند.

۲٫۲٫۲ تابع void LetMeBe(PlayerSign)

بازی‌کن `me*` را، پس از تعیین این‌که آیا برنامه باید شروع‌کننده باشد یا حریف وی، مقداردهی می‌کند.

۲٫۲٫۳ تابع void Initialize()

تصادفی(تر) بودن اعداد تولید شده توسّط تابع `rand()` را تثبیت کرده و نیز `me*` را برابر `O` (نفر دوم) قرار می‌دهد. در غیر این‌صورت، در صورتی‌که `me*` می‌بایست آغازگر باشد، در همان حین به `X` تبدیل می‌شود.

۲٫۲٫۴ تابع void ReadConf(char *)

از فایلی مشابه فایل اجرایی و با پسوند `.conf`، هشت پارامتر `COST_...` را می‌خواند. این تابع در کار با الگوریتم ژنتیک برای تعویض بازی‌کن صرفاً با استفاده از دادن این پارامترها در یک فایل خارجی، مورد استفاده واقع شده‌است.

از این تابع در ارسال نهایی استفاده‌ای نشده است.

۲٫۳ توابعِ عضو کلاس Board

به‌جز توابع ساده قسمت قبل که در `main` برنامه فراخوانی می‌شوند، سایر توابع مهم و اصلی که توابع‌عضو کلاس `Board` می‌باشند، به شرح ذیل‌اند.

در این توابع، تا جای ممکن «ارسال با ارجاع»[20] صورت گرفته‌ست تا علاوه بر امکان تغییر پارامتر ارسالی در درون تابع (که برای اشیاء کلاس `Error` الزامی‌ست)، سرعت اجرای برنامه (با جلوگیری از کپی‌کردن اشیاء حجیم به‌هنگام فراخوانی توابع) نیز بهبود بخشیده‌شود.

۲٫۳٫۱ تابع bool CheckOneMove(Error &, Move, Player &)

که با بررسی‌های لازم، تعیین می‌کند که آیا حرکت ارسال‌شده، قابلیت انجام شدن را، روی تخته‌ی متعلّق به آن (`this`) و توسّط بازیکن ارسال شده، دارد یا خیر؟ این تابع، از مقدار عددی یک عنصر تاس بی‌اطلاع بوده و

---

[19] Standard Error (cerr)
[20] Call By Reference

بررسی این مهم که اندازه‌ی حرکت با اندازه‌ی تاس برابرست یا خیر، خارج از وظایف آن است. به‌طور دقیق‌تر، از آن‌جا که کلیه‌ی حرکات مجاز تولید (و نه بررسی) می‌شوند، این امر نیازی به بررسی شدن مجزّا ندارد.

۲٫۳٫۲ تابع bool CheckInst(Error &, Instruction &, Dice &, Player &)

که عمل بررسی صحّت عملیات را با گرفتن یک تاس انجام می‌دهد. در ابتدای این‌تابع، کلیه‌ی حرکات مجاز و بیشینه (از نظر تعداد) با استفاده از تاس ارسالی، توسّط بازیکن ارسالی و روی تخته‌ی متعلّق به آن (this)، توسّط تابع MaximalMoves() تولید می‌شود. پس از آن، ابتدا تعداد حرکات ارسالی (اندازه‌ی آرایه‌ی پویای Instruction) با حرکات ماکسیمال، مقایسه می‌شود و سپس به‌ازای هر جای‌گشت از حرکات ورودی، بررسی می‌کند که آیا این جای‌گشت برابر یکی از دنباله حرکات تولید شده توسّط MaximalMoves() هست یا خیر؟

برای تولید جای‌گشت‌های دنباله‌حرکت (Instruction) ورودی، از تابع next_permutaion() که از توابع سودمند «کتاب‌خانه‌ی استاندارد قالبی»[۲۱] زبان C++ است استفاده شده است.

۲٫۳٫۳ تابع int FarthestCanEat(Player &)

این تابع از یک‌سو بررسی می‌کند که آیا بازی‌کن ارسالی، در تخته‌ی متعلّق به آن (this)، امکان خوردن دارد یا خیر، و از سوی دیگر، اگر جواب مثبت بود، دورترین خانه‌ی مهره‌دار برای خوردن را برمی‌گرداند.

این مهم با شماردن خانه به خانه‌ی تعداد مهره‌های موجود در سطر خانگی (۶ خانه‌ای که مهره‌ها صرفاً از آن‌ها خورده می‌شوند) به‌دست می‌آید.

۲٫۳٫۴ تابع void DFSMoves(InsPack &, Instruction &, unsigned, int, Dice&, Player &)

این تابع که به‌صورت بازگشتی (DFS-وار) فراخوانی می‌شود، وظیفه‌ی اصلی تولید دنباله حرکات بیشینه را دارد. برای جلوگیری از تولید حالات تکراری، مقدار ستونی که باید در فراخوانی بعدی جست‌وجو از آن شروع شود ارسال می‌شود. این پارامتر، در صورتی که حرکت قبلی، نشانیدن یک مهره‌ی زده شده باشد، برابر محل شروع مهره‌های بازی‌کن و در غیر این‌صورت برابر با خانه‌ی کنونی یا خانه‌ی بعد از کنونی خواهد بود.

۲٫۳٫۵ تابع InsPack MaximalMoves(Dice &, Player &)

شاید بتوان این تابع را قلب تولید حرکات دانست! این تابع، یک آرایه(ی پویا) از دنباله حرکت (Instruction) بر می‌گرداند که در آن کلیه‌ی حرکاتی که از تعدادی بیشینه هستند و بازی‌کن ارسال توسّط تاس ارسالی روی تخته‌ی کنونی می‌تواند انجام دهد، جمع‌آوری شده است.

با کمک این تابع، تمام فرزندان یک رأس از درخت بازی با استفاده از یال‌های برگردانده شده توسّط این تابع می‌توانند ساخته شوند.

در این تابع، فراخوانی اوّلین عمق DFSMoves() به‌ازای تمامی جای‌گشت‌های ممکن از تاس‌ها (که حداکثر دو جای‌گشت متفاوت هستند!) صورت می‌گیرد. این جای‌گشت‌ها به‌سهولت توسّط تابع next_permutation ساخته می‌شوند.

در پایان نیز، دنباله حرکت‌های ساخته‌شده مرتّب[۲۲] و یکتا[۲۳] می‌شوند.

---

[21] Standard Template Libarary (STL)
[22] Sort

۲٫۳٫۶ تابع void DoOneMove(Move &)

این تابع حرکات ارسال شده را روی تخته‌ی متعلّق به آن انجام می‌دهد. فرض می‌شود این حرکت قبلاً بررسی شده و نتیجتاً ممکن و معقول است.

۲٫۳٫۷ تابع void DoInst(Instruction &)

یک دنباله از حرکات را یک‌به‌یک و به‌ترتیب داده شده توسّط `DoOneMove` انجام می‌دهد. از آن‌جا که این تابع، هیچ بررسی‌ای روی صحّت ترتیب حرکات انجام نمی‌دهد، لازم‌ست تا دنباله‌حرکات (`Instruction`) ارسالی، قبلاً از نظر ترتیب وقوع ممکن تثبیت شوند که این مهم از وظایف جانبی `CheckInst` است.

۲٫۳٫۸ تابع PlayerSign Winner()

در صورتی که تمام مهره‌های یکی از دو بازی‌کن خورده شده باشد، `sign` آن بازی‌کن و در غیر این‌صورت، `sign` بازی‌کنِ `NOP` را بر می‌گرداند. این تابع برای تشخیص اتمام بازی سودمند است.

۲٫۳٫۹ تابع bool HasConflict()

تعیین می‌کند که آیا بازه‌های حرکتی دو بازیکن با هم تلاقی دارد یا خیر؟ به‌عبارت دیگر، آیا امکان دارد در حالتی در آینده، مهره‌ای زده شود؟

می‌توان توجیه کرد که در صورت عدم تلاقی، هیچ‌یک از دو بازی‌کن لزومی به نگرانی برای لَت (ستون تک‌مهره‌ای) های خود ندارند.

۲٫۳٫۱۰ تابع Instruction ChooseMyMoves(Dice, Player *, int, double, double, double)

مغز اصلی بازی، این تابع است! با استفاده از مقادیر `alpha` و `beta`ی ارسال شده به‌عنوان پارامترهای پنجم و ششم، این تابع حرکتی را که بازی‌کن ارسالی با تاس ارسالی از تخته‌ی متعلّق به‌آن (که در عمقِ ارسالی (توسّط پارامتر سوم) قرار دارد)، به‌تر است انجام دهد برمی‌گرداند.

در پیاده‌سازی این تابع، تکنیک‌های ExpectiMinimax و α-β به‌کار گرفته شده‌اند. برگ‌های درخت، پس از رسیدن به عمق خاص (که در ثابت `MINMAX_DEPTH` لحاظ شده و در فراخوانی اوّلیه از `main` ارسال می‌شود)، توسّط تابع `Eval()` ارزیابی می‌شوند.

۲٫۳٫۱۱ تابع double Eval(int, Player *, int, double, double)

تابع مکمّل α-β است، که در صورتی که عمق درخت برابر `MINMAX_DEPTH` باشد، آن را با کمک ثابت‌های مهم `COST_...` ارزیابی کرده و تبدیل به یک عدد اعشاری می‌کند. در غیر این‌صورت، الگوریتم ExpectiMinimax بر روی فرزندان اِعمال می‌شود. شیوه‌ی اجرای این الگوریتم چنین‌ست که وزن یک رأس برابرست با وزن به‌ترین حرکت (بسته به بازی‌کن، دارای بیش‌ترین یا کم‌ترین ارزش) هر یک از فرزندان (به ازای هر یک از ۲۱ تاس ممکن) ضرب‌در احتمال آمدن آن تاس.

---

[23] Unique

هر چه‌قدر خروجی این تابع بیش‌تر باشد، به‌ضرر بازی‌کن X و هر چه‌قدر کم‌تر (منفی) باشد، به‌ضرر O است. از این‌رو بازی‌کن X در تابع `ChooseMyMoves` سعی می‌کند به گره‌ی با کم‌ترین ارزش (ترجیحاً منفی با قدرمطلق زیاد) برود و بالعکس O در تلاش برای رفتن به گره‌ی پرارزش‌تر است.

۲٫۳٫۱۲ تابع `int Eval(Player &)`

امتیاز بازی‌کن ارسال شده در پایان بازی را حساب می‌کند. بسته به باخت، برد یا مارس کردن[۲۴] حریف، این امتیاز ممکن‌ست صفر، یک یا دو باشد. چنان‌که پیش‌تر گفته شد، امتیاز ۳ برای حالت backgammon در نظر گرفته نشده است.

۲٫۴ متغیّرهای عمومی[۲۵] برنامه

به‌جز مقادیر ثابت `COST_...` که از الگوریتم ژنتیک به‌دست آمده و در تابع ارزیابی[۲۶] به‌کارگرفته‌می‌شوند، تنها ۴ متغیّر عمومی در برنامه وجود دارد:

- آرایه‌ی `player` که برای بازی‌کن‌های X و O اهمیّت داشته و فراخوانی متدهای آن، بیش‌تر جنبه‌ی `static` دارند.
- `me` و `he` که اشاره‌گرهایی به بازی‌کنی که برنامه قرارست حرکات آن‌را انجام دهد و حریف وی دارند.
- `b`، تخته‌ی جاری و اصلی که تنها در تابع `main` و دقیقاً به‌هنگام حرکات اصلی تغییر می‌کند.

[24] Gammon
[25] Global
[26] Evaluation Function

۳ نتایج ضمنی؛ پارامتریزه‌کردن نرّاد

۳٫۱ اجرای اوّلیه

پس از اتمام مقدّمات برنامه‌سازی و رفع اشکلات فنّی[۲۷] موجود اوّلین بازی نرّاد انجام شد. در ابتدای امر، عمق Minimax برابر ۳ عمق تعیین شده‌بود که به‌دلیل اتلاف بسیار زیاد زمان، این مهم به ۲ کاهش یافت. مشاهده شد که حتی با اِعمال روش α-β نیز، به‌دلیل عمق کم پیمایش، عامل‌انشعاب[۲۸] بالا و احتمالی بودن بیش‌از حد رئوس اوّلیه، تغییر محسوسی در زمان هر حرکت رخ داد و این زمان تقریباً از ۳۰ ثانیه در هر حرکت به ۲۰ ثانیه رسید که برای بازی تخته‌نرد که در آن بعضاً تا ۱۰۰ حرکت در یک بازی انجام می‌شود، بسیار زیاد بود.

از سوی دیگر، به‌دلیل تصادفی بودن حرکات (وجود عامل تاس)، احتمال برد قطعی یک بازی‌کن *عالی* مقابل یک بازی‌کن *نسبتاً خوب* در این بازی، به‌طور معمول حداکثر ٪۷۵ است و از این رو، پیش‌رفت کار الزاماً یک‌سویه و «مستقیم به هدف» نبود. این مشکل وقتی پررنگ‌تر می‌شود که تراگذر[۲۹] بودن رابطه‌ی بُرد، به احتمال قابل ملاحظه‌ای زیرسؤال رفته و شرایط برای نتیجه‌گیری قدرت یک بازی‌کن بحرانی‌تر شود.

۳٫۲ پارامترهای نرّاد

با تغییر ثابت‌های موجود در تابع ارزیابی برگ‌های درخت Minimax، کاشف به‌عمل آمد که دو مقداردهی معقول و نسبتاً نزدیک، می‌توانند نتایج سنگینی را بر یکی تحمیل کنند. با این وصف و تحلیل بیش‌تر پارامترهای مهم برای ارزیابی ایستا[۳۰]ی یک جدول، تعداد پارامترهای `COST_...` در تابع `Eval()` افزایش یافتند تا این تفاوت‌های ناچیز، محسوس‌تر و قابل تحقیق‌تر باشند.

پارامترهای سودمند و نهایی برای یک نرّاد به ۸ پارامتر زیر محدود شده‌اند. بالطبّع ممکن‌ست برخی از این پارامترها سودمند نباشند و یا پارامترهای دیگری بتوانند جای‌گزین شوند؛ که البته این استدلال، با تحلیل ضمنی بیش‌تر روی پارامترها کم‌رنگ می‌شود.

مجدداً ذکر می‌شود که خروجی تابع `Eval()` یک مقدار اعشاری‌ست که هر چه‌قدر بیش‌تر (مثبت‌تر) باشد، به‌ضرر X و هرچه‌قدر کم‌تر(منفی‌تر) باشد به ضرر O است. با این وصف، واضح‌ست که این پارامترها برای مهره‌های بازی‌کن X با ضریب ۱+ و برای مهره‌های بازی‌کن O با ضریب ۱- محاسبه شده و این ۲ مقدار با هم جمع می‌شوند.

قابل ذکرست که ۴ پارامتر اوّل، هزینه‌ی وقوع شرایط بد برای یک بازی‌کن (مهره‌ی روی بار داشتن، لَت دادن و ...) را توصیف می‌کند؛ حال آن‌که پارامترهای ۵ تا ۸ پارامترهای خوب بوده و شرایط خوب را، با دادن هزینه‌های منفی، معنادار می‌کنند. ضمناً پارامترهای سوم، چهارم و هشتم صرفاً ضرایبی برای سایر پارامترها (به‌ترتیب دوم، دوم و هفتم) می‌باشند که می‌توانند با تغییرات کوچک، اثرات بزرگ داشته‌باشند.

نهایتاً حدس اوّلیه‌ی و مقدار بهینه‌ی پیداشده‌ی نهایی (توسّط الگوریتم ژنتیک)، برای هر پارامتر در جلوی آن نوشته‌شده است. این پیش‌نهاد (حدس)های اوّلیه صرفاً بر اساس یک شهود حسّی تخمین زده شده و هیچ ملاک دقیقی ندارند.

---

[27] bugs
[28] Branching Factor
[29] Transitive
[30] Static

۳٫۲٫۱ پارامتر ثابت؛ فاصله‌ی منهتن

عاملی که به عنوان واحد در نظر گرفته شده است تا ضرب تمامی پارامترها در یک عدد ثابت بی‌اثر نبوده (و نتیجتاً جواب بهینه تقریباً یک‌تا شود)، مجموع فواصلی است که تمامی مهره‌ها باید طی کنند تا خورده شده و وارد منزل شوند. به‌عبارت دیگر، مستقل از مهره‌های حریف چند تاس یک‌دانه‌ای عدد یک لازم‌ست تا بازی‌کن تمام مهره‌هایش را از دست بدهد. برای مهره‌ی روی بار، این فاصله طبق تعریف برابر ۲۵ درنظر گرفته می‌شود.

۳٫۲٫۲ پارامتر اوّل: COST_ON_BAR {پیش‌نهاد: ۸۰، نهایی: ۱۸۲٫۸}

این پارامتر، هزینه‌ی وجود یک مهره روی بار را بیان می‌کند. بالطّبع یک مهره‌ی زده‌شده توسّط حریف (که روی بار قرار دارد) در فاصله‌ی منهتن نیز مؤثر است؛ الزامی ندارد که این تأثیر مابین مهره‌ی روی بار و ستون ۷ واحد جلوتر از بار (که با آمدن ۴ و۳ در شرایط مطلوب مهره‌ی روی بار به آن ستون می‌رود) برابر ۷ واحد باشد؛ چرا که نشستن (بازگشت به بازی) برای یک مهره‌ی زده شده می‌تواند سخت‌تر بوده و موجب از دست دادن چندین تاس شود.

۳٫۲٫۳ پارامتر دوم: COST_BLOT {پیش‌نهاد: ۲۰، نهایی: ۷}

هزینه‌ی داشتن یک لَت[31] (ستونی که تنها یک مهره دارد).

با استفاده از تابع HasConflict() در صورتی که مهره‌های دو بازیکن کاملاً یک‌دیگر را پشت‌سر گذارده باشند و امکان هیچ‌گونه زدن‌ای در آینده نباشد، اثر این پارامتر (مطابق خط ۴۷۹)، خنثی می‌شود.

۳٫۲٫۴ پارامتر سوم: COST_BLOT_HR {پیش‌نهاد: ۱٫۵، نهایی: ۱}

این پارامتر ضریبی برای پارامتر دوم است، در حالتی که لَت در یکی از ۶ خانه‌ی ابتدایی (جایی که حریف زده شده با یک تاس می‌تواند بنشیند) باشد. بالطّبع وجود چنین لَتی، به‌دلیل اِعمال ترس از زدن حریف (که می‌تواند زده شده بازی‌کن را بعد از نشستن حریف درپی داشته باشد)، خطرناک است. نیز اگر این لَت زده شده و به بار منتقل شود، فاصله‌ی منهتی قابل ملاحظه‌ای را اجبار می‌کند.

۳٫۲٫۵ پارامتر چهارم: COST_PRESENCE {پیش‌نهاد: ۲، نهایی: ۴٫۱۵}

آخرین پارامتر هزینه که مشابه پارامتر سوم، ضریبی برای پارامتر دوم است، اثر خطرناک بودن یک لَت نسبت به موقعیّت مهره‌های حریف را محاسبه می‌کند. واضح‌ست که یک لَت ممکن‌ست در جایی با یک تاس (حداکثر ۶ ستون فاصله) زده نشود حال آن‌که در جایی دیگر، در همان نزدیکی یک لَت در برابر تعداد زیادی دیوار حریف قرار داشته باشد و تقریباً با هر تاسی قابل زده شدن باشد!

این پارامتر، چنان که در سطور ۴۸۰ تا ۴۸۵ محاسبه و در سطر ۴۸۹ اِعمال شده، به‌طور مستقیم و عددی در ضریب لَت ضرب می‌شود.

۳٫۲٫۶ پارامتر پنجم: COST_CONNECT {پیش‌نهاد: ۱۰-، نهایی: ۴۷٫۸-}

اوّلین پارامتر خوب که هزینه‌ها را کم می‌کند، پیوستگی دیوارها را بیان می‌کند. یک دیوار، ستونی از بازی‌کن است که در آن دو یا بیش‌تر مهره وجود داشته باشد و نهایتاً از حرکت حریف جلوگیری به‌عمل بیاورد. روشن‌ست که اگر دو یا چند دیوار به‌هم متّصل باشند، شانس عبور حریف را به‌صورت نمایی کاهش می‌دهند.

---

[31] Blot

این پارامتر در صورتی که یک دیوار مجاور یک دیوار دیگر باشد، در هزینه‌ی منفی (خوب بودن) آن دیوار ضرب می‌شود.

۳٫۲٫۷ پارامتر ششم: COST_EATEN {پیش‌نهاد: ۱۰-، نهایی: ۷۸٫۶-}

خوردن یک مهره (به منزل)، هدف غایی بازی است. این پارامتر، میزان خوبیّت یک مهره‌ی خورده شده را تعیین می‌کند. واضح‌ست که سنگین بودن وزن این پارامتر ممکن‌ست لَت‌دادن در شرایط بحرانی را خنثی کند.

۳٫۲٫۸ پارامتر هفتم: COST_WALL {پیش‌نهاد: ۱۰-، نهایی: ۱۰-}

هزینه‌ی منفیِ ساختن دیوار.

۳٫۲٫۹ پارامتر هشتم: COST_WALL_HR {پیش‌نهاد: ۳، نهایی: ۵٫۷۹۷}

هزینه‌ی منفی ساختن یک دیوار در ۶ ستون خانگی که از نشستن (ورود) مهره‌ی زده‌شده (روی بار) حریف ممانعت به‌عمل آورده و حتی ممکن‌ست یک دست بازی (تاس) از حریف را از بین ببرد!

۴ الگوریتم ژنتیک

۴٫۱ مشکل راه‌اندازی بازی

مهم‌ترین مشکل در زمینه‌ی اجرای الگوریتم ژنتیک، دشواری برقراری بازی بین دو بازی‌کن بود که جریان ورودی یکی را به‌عنوان خروجی دیگری (و بالعکس) تلقّی کند. این مهم در زبان Java توسّط خادم نوشته شده و با کمک Socket Programming رفع شد اما جای‌گزین مناسب برای اتّصال یک برنامه‌ی C++ به یک Socket احتیاج به کتاب‌خانه[۳۲]های خاص خود داشت.

نهایتاً با کمک ابزار Caia که قاضی رسمی مسابقات بازی‌کننده‌های سایت http://www.codecup.nl است و بازخوانی کد آن و نهایتاً تبدیل داور و برنامه‌ریز آن به فُرمت‌های مناسب تخته‌نرد، این معضل حل شد و امکان برقراری بازی بین دو بازی‌کن (۲ فایل `.exe`) برقرار شد.

فایل‌های `manager.cc`، `manager.exe`، `manager.txt`، `referee.cc`، `referee.exe`، `ca.exe` و نیز پوشه‌های `playerlogs` و `refereelogs` موجود در بسته‌ی ارسالی مربوط به این ابزارها می‌باشند. برای اجرای آن، کافی‌ست ۲ برنامه با نام‌های `player1.exe` و `player2.exe` موجود بوده و فایل `ca.exe` اجرا شود. توضیحات بیش‌تر درباره‌ی نحوه‌ی اتّصال و گفت‌وگوی manager و referee در سایت مربوطه آمده است.

۴٫۲ مشکل دوم، زمان

به‌دلیل وجود عامل تصادفی‌بودن (یا *شانس*) در این بازی، بین هر دو بازیکن در هر دست بازی، به‌تر می‌بود که بیش از یک بازی انجام شود تا نتیجه حدالمقدور به مقدار واقعی نزدیک باشد. از سوی دیگر، از آن‌جا که هر دست بازی حدود ۸ ثانیه طول می‌کشید، باید حداقل صرفه‌جویی در زمان می‌شد. با این وصف، قرار بر این شد که در هر بازی بین دو بازی‌کن، ۵ دست (دفعه) بازی انجام شود و نتیجه‌ی هر دو بازی‌کن ذخیره و جمع شود. نتیجتاً هر بازی دو نفره (که ۵ دست خواهد داشت) حدود ۴۰ ثانیه به‌طول می‌انجامد.

با بررسی دقیق‌تر، تصمیم گرفته‌شد که در هر نسل از جمعیّت یک تورنمنت کامل[۳۳] بین افراد زنده برگزار شود تا عامل شانس از این پررنگ‌تر نشود. مجدداً، به دلیل همین محدودیت زمانی، اندازه‌ی جمعیّت فعّال در هر نسل برابر ۲۰ نفر تعیین شد تا هر نسل حدود ۷۰ الی ۹۰ وقت بگیرد.

واضح‌ست که در صورت وجود پردازنده‌های خاص و قوی‌تر و نیز محدودیت کم‌تر زمانی، این جمعیّت فعّال می‌توانست افزایش یافته و نیز تعداد بازی‌های بیش‌تری مابین هر دو بازی‌کن انجام شود.

۴٫۳ انتخاب نسل اوّلیه، جمعیّت، نرخ رشد

برای نسل اوّلیه (نسل صفر)، بازی‌کن حدسی (که پارامترهای پیش‌نهاد شده‌ی آن نوشته شده است) خوب بود که در جمعیّت باشد. از سوی دیگر، برای جلوگیری از خروج بی‌رویه و بی‌معنی پارامترها لازم بود که برای هر پارامتر یک بیشینه و کمینه تعریف شود. با تفکّر روی این مهم، نهایتاً تصمیم بر این شد تا در نسل صفر، دو بازی‌کن با

---

[32] Library
[33] Full Tournament

کمینه‌ی پارامترها و بیشینه‌ی پارامترها نیز گنجانده شوند. این دو بازی‌کن صرفاً به تنوّع در هم‌گذری[۳۴] در ابتدای کار کمک می‌کنند و لزوماً بازی‌کن‌های مطلوبی نیستند. جالب توجه‌ست که به‌دلیل وجود پارامترهای خوب و بد، نتیجه‌ی بازی بین بازی‌کن دارای کمینه‌ی پارامترها و بازی‌کن دارای بیشینه‌ی پارامترها قابل پیش‌بینی نبود!

اما برای تولید نسل، تصمیم بر این شد که اوّلاً از سیستم «نخبه‌پروری[۳۵]» استفاده شود. علی‌رغم مضرّات این سیستم نظیر جلوگیری از تنوّع و ...، به‌دلیل حجم کوچک جمعیّت و تعداد کم نسل‌ها، این سیستم به‌طور مطلوب‌تری به یک نقطه‌ی نسبتاً ثابت (بازی‌کن خوب‌تر، Local Maxima) میل می‌کند.

با این وصف، در پایان هر نسل، ۱۰ بازی‌کن دارای بیش‌ترین امتیازات از تورنمنت کامل انتخاب شده و ۱۰ بازی‌کن ضعیف‌تر کشته می‌شوند. سپس با استفاده از ۱۰ بازی‌کن قوی زنده‌مانده، ۵ کودک به‌صورت هم‌گذری و ۵ کودک به‌صورت جهش به نسل اضافه می‌شدند تا مجدداً یک تورمنت برگزار شود. در «هم‌گذری»، دو والد تصادفی از ۱۰ بازی‌کن قوی نسل قبل انتخاب شده و هر یک از ۸ پارامتر کودک به‌طور تصادفی و با احتمال یک‌سان از یکی از دو والد انتخاب می‌شود. در جهش نیز، یکی از افراد قوی انتخاب شده و کودک وی، همان فرد است که یکی از ۸ پارامتر آن به یک مقدار تصادفی در بازه‌ی معقول جهیده است.

ضمناً از آن‌جا که ۱۰ بازی‌کن هر نسل، در نسل بعد تکراری هستند، بازی‌های مابین این بازی‌کنان در یک `map<pair <Player, Player>, pair<int, int> >` ذخیره می‌شود، تا از انجام بازی تکراری جلوگیری کرده و نهایتاً در زمان صرفه‌جویی شود.

با این معیّنات، در مدّت حدود ۶۰ ساعت بازی بی‌وقفه، الگوریتم برای ۵۰ نسل متوالی اجرا شد که نتایج آن در فصل بعد گردآوری شده است.

---

[34] Cross Over
[35] Elitism

۵ نتیجه‌ی الگوریتم ژنتیک

پس از ۵۰ نسل بازی، یکی از بازی‌کنان توانست در نسل ۵۰اُم با گرفتن ۸۰ امتیاز در ۱۹ بازی (میانگین ۴٫۲۱ در هر سِت ۵تایی) در رتبه‌ی اوّل قرار گیرد. در تمجید از این بازی‌کن همین بس که وی توانست در ۳ نسل بعدی نیز همواره رتبه‌ی اوّل را به‌خود اختصاص دهد و فرزند دقیقاً مشابه وی (که با هم‌گذری از وی و یکی دیگر از بازی‌کنان شبیه به آن تولید شده) توانست در نسل ۵۱ اُم رتبه‌ی دوم را به خود اختصاص دهد.

از سوی دیگر، در پایان کار، بازی‌کن قهرمان نسل‌ای ۵۰ تا ۵۳ همواره در مقابل بازی‌کن حدس اوّلیه (از نسل صفر) باز را با نتیجه‌ی سنگین ۶ بر ۱ در هم شکست.

رتبه‌بندی نهایی بازی‌کنان در پایان هر نسل در فایل `ga.log` آمده است.

تحلیل این نتایج، و بحث چگونگی تثبیت یا نوسانات پارامترها، طول عمر بازی‌کنان و ... خارج از حیطه‌ی این پروژه است.

«پایان»