حل مسئله N وزیر و مقایسه الگوریتم های اصلی

عبداله کشتکار

درس هوش مصنوعی

استاد مجتبایی

دانشجوی مهندسی تکنولوژی نرم افزار

دانشگاه فنی حرفه ای مهاجر اصفهان

چکیده

مسئله چند وزیر یک معمای شطرنج و ریاضیاتی است که بر اساس آن باید N وزیر را در یک صفحه N*N شطرنج قرار داد، جایگاه این N وزیر باید به صورتی باشد که همدیگر رو تهدید نکنند.

مقدمه

معروفترین شکل این مسئله معمای هشت وزیر است که برای حل آن باید 8 وزیر را در صفحه 8 در 8 قرار داد، این مسئله 92 جواب دارد که 12تای آنها منحصر به فرد است، یعنی بقیه جواب ها از تقارن جواب های اصلی بدست می آیند.

ما در اینجا این مسئله را با روش عقب گرد حل کردیم سپس الگوریتم مونت کارو را پیاده کردیم و درآخر الگوریتم ژنتیک را نوشتیم و مقایسه بین این الگوریتم و الگوریتم عقب گرد کردیم.

فهرست

تاریخچه

این مسئله در سال 1848 توسط شطرنج بازی به نام Max Bezzel عنوان شد و ریاضی دانان بسیاری از جمله Gauss و Georg Cantor بر روی این مسئله کار کرده و درنهایت آن را به N وزیر تعمیم دادند.

اولین راه حل توسط Franz Nauck در سال 1850 ارائه شد که به همان مسئله N وزیر تعمیم داده شد، پس از آن Gunther راه حلی با استفاده از دترمینان ارائه داد که J.W.L Glaisher آن را کامل نمود. در سال 1979 اقای Edsger Dijkstra Nauck این مسئله را با استفاده از عقب گرد حل کرد.

صورت مسئله

مسئله N وزیر درصورتی جواب دارد که N برابر 1 یا بزرگتر سه باشد، به این معنی است که ما برای دو و سه وزیر جواب نداریم.

مسئله N وزیر از جمله مسائل NP در هوش مصنوعی می باشد که با روش های جستجوی معمولی قابل حل نیست.

حالت شش وزیر جواب های کمتری نسبت به حالت پنج وزیر دارد و فرمول صریحی برای یافتن تعداد جواب ها وجود ندارد.

تعداد جواب ها

تعداد وزیر ها	1	2	3	4	5	6	7	8	9
منحصر به فرد	1	0	0	1	2	1	6	12	46
متمایز	1	0	0	2	10	4	40	92	352

روش های حل مسئله

عقب گرد | Backtracking

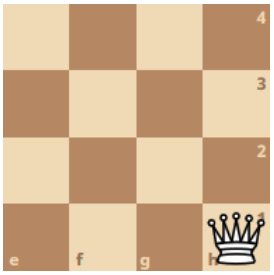
از تکنیک عقب گرد برای حل مسائلی استفاده می شود که در آنها دنباله ای از اشیا از یک مجموعه مشخص انتخاب می شود، به طوری که این دنباله ملاکی را در بر دارد. عقبگرد حالت اصلاح شده جستجوی عمقی یک درخت است. این الگوریتم همانند جستجوی عمقی است، با این تفاوت که فرزندان یک گره فقط هنگامی ملاقات می شوند که گره امید بخش باشد و در آن گره حلی نباشد.

به فرض اینکه وزیر در خانه [I,J] قرار داشته باشد، مهره های که در خانه های (I,M)، (J,M) یا (I±M, J±M) قرار دارند توسط وزیر تهدید می شوند.

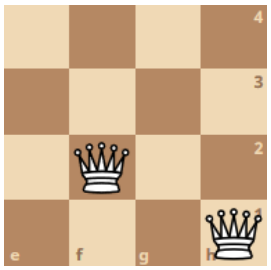
برای سادگی تشریح مسئله با استفاده از روش عقبگرد فرض میکنیم که که خانه های شطرنج 4 در 4 و تعداد وزیر ها نیز 4 باشد.

مراحل جستجو برای یافتن جواب را به این صورت دنبال می کنیم که:

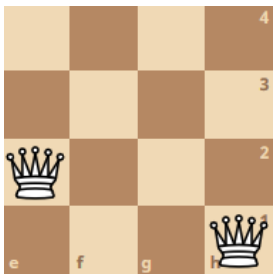
1. **وزیر اول را در ردیف اول و ستون اول قرار میدهیم**



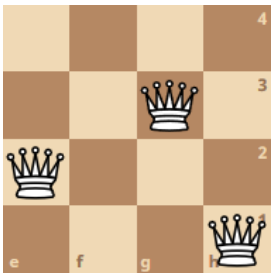
2. **در ردیف دوم از اولین ستون به جلو رفته و به دنبال خانه ای میگردیم که مورد تهدید وزیر اول نباشد و وزیر دوم در آن جا قرار میدهیم**



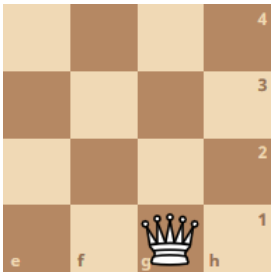
3. **همانند قبل، در ردیف سوم از اولین ستون به جلو رفته و به دنبال خانه‌ای می‌گردیم که مورد تهدید وزیران اول و دوم نباشد. می‌بینیم که چنین خانه‌ای موجود نیست. پس به عقب یعنی ردیف دوم برگشته و وزیر دوم را به خانه‌ای دیگر از ردیف دوم منتقل می‌کنیم که مورد تهدید وزیر اول نباشد.**



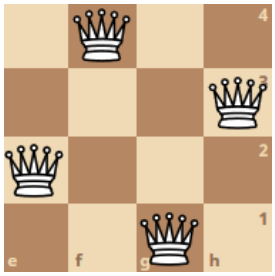
4. دوباره در ردیف سوم اولین خانه‌ای را میابیم که مورد تهدید دو وزیر قبلی نباشد. این بار خانه را می‌یابیم و وزیر سوم را در آن قرار می‌دهیم.



5. همانند قبل، در ردیف چهارم به دنبال اولین خانه‌ای می‌گردیم که مورد تهدید وزیران پیشین نباشد. چنین خانه‌ای موجود نیست. به ردیف قبل یعنی ردیف سوم باز می‌گردیم تا خانه‌ای دیگر برای وزیر سوم بیابیم. خانه دیگری وجود ندارد. به ردیف قبل یعنی ردیف دوم بر می‌گردیم تا خانه دیگری برای وزیر دوم پیدا کنیم. به آخرین ستون رسیده‌ایم و خانه دیگری نیست. به ردیف قبل یعنی ردیف اول بر می‌گردیم و وزیر اول را یک ستون به جلو می‌بریم.



6. در ردیف دوم اولین خانه‌ای را میابیم که مورد تهدید وزیر اول نباشد و وزیر دوم را در آن خانه قرار می‌دهیم.
7. در ردیف سوم اولین خانه‌ای را میابیم که مورد تهدید وزیران اول و دوم نباشد و وزیر سوم را در آن خانه می‌گذاریم.
8. در ردیف چهارم اولین خانه‌ای را میابیم که مورد تهدید وزیران پیشین نباشد. این بار خانه را می‌یابیم و وزیر چهارم را در آن خانه قرار می‌دهیم.



9. به یک جواب می‌رسیم. حال اگر فرض کنیم که این خانه جواب نیست و به مسیر خود ادامه دهیم، احتمالاً" می‌توانیم جوابهای دیگری نیز بیابیم.

شبه کد پیاده سازی الگوریتم عقبگرد برای مسئله N وزیر

```
Void queens(index i) {
        Index j;
        If (promising(i))
                If (i==n)
                        Cout << col[1] through col[n];
                Else
                        For(j=1; j<=n; j++){
                        Col[i+1]=j;
                        Queens(i+1);
                }
}
Bool promising(index i) {
        Index k;
Bool switch;
k == 1;
Switch = true;
While(k < I && switch)
{
        If (col[i] == col[k] || abs(col[i] – col[k] == i – k)
                Switch = false;
        k++;
}
}
```

کد پایتون

شما میتوانید از پروژه با Import کردن کلاس NQueenBackTracking از فایل backtracking و صدا زدن فانکشن run عمل عقب گرد و حل مسئله را مشاهده نمایید.

الگوریتم مونت کارلو

از الگوریتم مونت کارلو برای برآورد کردن کارایی یک الگوریتم عقبگرد استفاده می‌شود. الگوریتم‌های مونت کارلو، احتمالی هستند، یعنی دستور اجرایی بعدی گاه به‌طور تصادفی تعیین می‌شوند. در الگوریتم قطعی چنین چیزی رخ نمی‌دهد. الگوریتم مونت کارلو مقدار مورد انتظار یک متغیر تصادفی را که روی یک فضای ساده تعریف می‌شود، با استفاده از مقدار میانگین آن روی نمونه تصادفی از فضای ساده بر آورد می‌کند. تضمینی وجود ندارد که این برآورد به مقدار مورد انتظار واقعی نزدیک باشد، ولی احتمال نزدیک شدن آن، با افزایش زمان در دسترس برای الگوریتم، افزایش می‌یابد.

کد پایتون

با استفاده از کلاس NQueenMonteCarloBackTracking **میتوانید کارایی الگوریتم عقبگرد را امتحان کنید.**

روش استفاده به این صورت است که با صدا زدن فانکشن montecarlo **با ورودی** numbers **که یک عدد صحیح و تعداد اجرای الگوریتم است به شما میانگین برمیگرداند.**

همچنین با صدا زدن فانکشن run **شما میتوانید نتیجه با تعداد اجرای** 1000**بار در کنسول مشاهده نمایید.**

الگوریتم ژنتیک

الگوریتم‌های ژنتیک یکی از الگوریتم‌های جستجوی تصادفی است که ایده آن برگرفته از طبیعت می‌باشد. الگوریتم‌های ژنتیک برای روش‌های کلاسیک بهینه سازی در حل مسائل خطی، محدب و برخی مشکلات مشابه بسیار موفق بوده‌اند ولی الگوریتم‌های ژنتیک برای حل مسائل گسسته و غیر خطی بسیار کاراتر می‌باشند. به عنوان مثال می‌توان به مسئله فروشنده دوره‌گرد اشاره کرد. در طبیعت از ترکیب کروموزوم‌های بهتر، نسل‌های بهتری پدید می‌آیند. در این بین گاهی اوقات جهش‌هایی نیز در کروموزوم‌ها روی می‌دهد که ممکن است باعث بهتر شدن نسل بعدی شوند. الگوریتم ژنتیک نیز با استفاده از این ایده اقدام به حل مسائل می‌کند.

در الگوریتم‌های ژنتیک ابتدا به‌طور تصادفی یا الگوریتمیک، چندین جواب برای مسئله تولید می‌کنیم. این مجموعه جواب را جمعیت اولیه می‌نامیم. هر جواب را یک کروموزوم می‌نامیم. سپس با استفاده از عملگرهای الگوریتم ژنتیک پس از انتخاب کروموزوم‌های بهتر، کروموزوم‌ها را باهم ترکیب کرده و جهشی در آنها ایجاد می‌کنیم. در نهایت نیز جمعیت فعلی را با جمعیت جدیدی که از ترکیب و جهش در کروموزوم‌ها حاصل می‌شود، ترکیب می‌کنیم. روند استفاده ما از الگوریتم‌های ژنتیک به صورت زیر می‌باشد:

معرفی جواب‌های مسئله به عنوان کروموزوم یا Generation

به عبارتی کروموزوم ها (همچنین نسل صدا زده می شوند) یک روشیه برای نشان دادن راه حل که ممکنه هم معتبر باشه یا نباشه، چطور یک جواب در این مسئله نشون بدیم؟ ساخت یک ارایه دو بعدی y .x با مقادیر 0 و سپس پرکردن آن با جایگاه وزرا، شاید این اولین پاسخی باشد که به ذهنتان بخورد ولی این راه حل مناسبی نیست، یک روش بهتر اینکه یک ارایه تک بعدی بسازیم سپس مقادیر بعنوان y و اندکس آن بعنوان x در نظر بگیریم. (لازم به ذکر است که مقادیر اندکس و مقادیر آرایه از 0 تا n-1 می باشد)

		Q	
Q			
			Q
	Q		

2	0	3	1

معرفی تابع برازندگی یا Utility

تابع برازندگی خوبی یک نسل را بررسی میکند، پس تابع یک کروموزوم به عنوان ورودی دریافت میکنند و مقدار برازندگی را برمیگرداند که درمورد ما تهدید های وزیر ها به هم این عدد می باشد به عبارتی اگه 4 تا وزیر هم دیگر رو تهدید کنند تعداد تهدید ها را برمیگرداند به این صورت است که میفهمیم که آیا این نسل برای تکثیر به درد میخورد یا خیر.

معرفی تابع تقاطع یا Crossover

همانطور که گفتیم کروموزوم ترکیب DNA دوتا والدینش است، به این عمل تقاطع می گویند، این فانکشن کلید مسئله است و کاری میکند که الگوریتم ژنتیک سریعتر عقب گرد باشد. ورودی این تابع دوتا کروموزوم است که قراره یک بچه تولید مثل کنند. اگر در اینترنت جستجو کنیم کلی تابع برای انجام این کار داریم.

برای الگوریتم ما براساس تحقیقات انجام شده اومدیم جای دو آیتم هر وقت اختلاف مقدار آنها کمتر 2 باشد جا به جا کردیم

معرفی تابع انتخاب یا Fight

در این تابع محیط جدید میسازیم، ولی قبل از آن با استفاده از تابع برازندگی بررسی میکنیم که ایا نتیجه هدف داریم یا خیر، اگه داشتیم نتیجه را برمیگردانیم در غیر اینصورت بهترین nتا نسل انتخاب میکنیم و درمحیط جدید قرار میدهیم. نحوه انتخاب برنده هم براساس عددی که تابع برازندگی به ما میدهد.

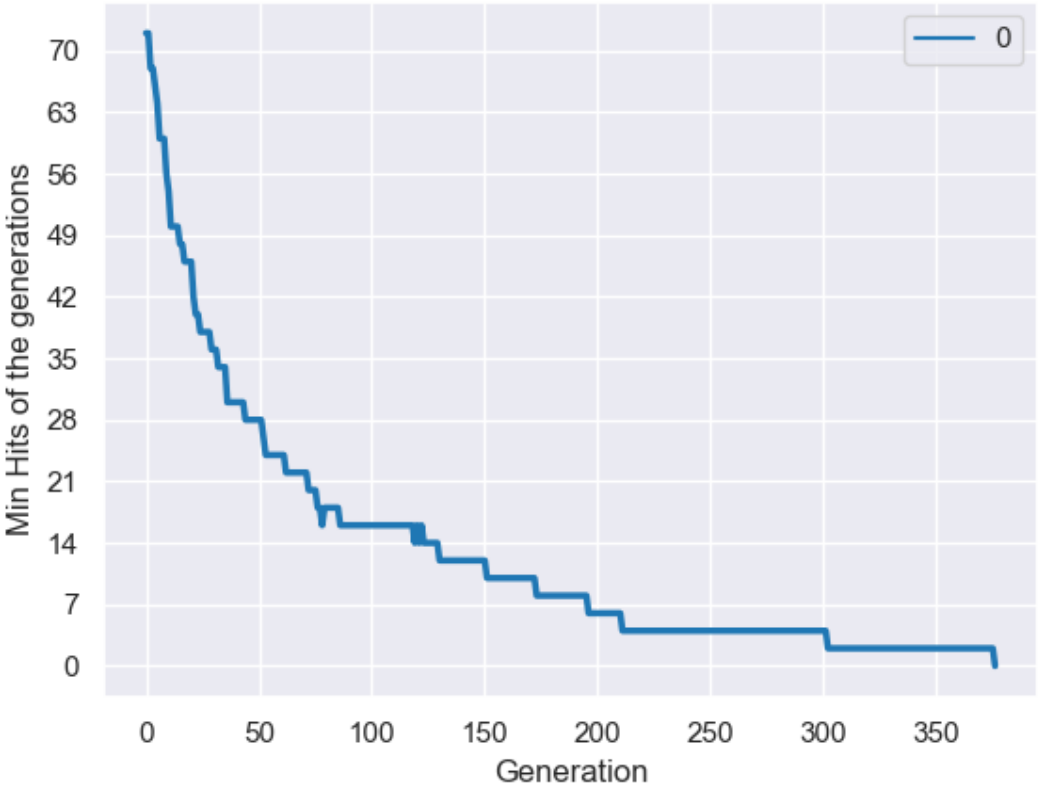
معرفی تابع جهش یا Mutation

این تابع یک نسل به عنوان ورودی میگیرد و به آن جهش میدهد که ما اول باید اعداد تکراری را حذف کنیم(برای کاهش ضربه ها)، سپس اعدادی که نیستند اضافه میکنیم، همچنین یک y از سمت راست آرایه با سمت چپ جا به جا میدهیم (تست)

کد پایتون

با صدا زدن کلاس NQueenGeneticAlgorithm سپس run میتوان نتیجه را مشاهده کرد(تابع convertor نتیجه را در دیکشنری queens کلاس قرار میدهد و میتوانید آن را چاپ کنید یا با صدا زدن set_up و finish میتوانید نتیجه را مشاهده نمایید)

نمودار پیشرفت در حالت 80 وزیر با استفاده از الگوریتم ژنتیک



در نمودار بالای مشاهده میکنید که با جلوتر رفتن نسل، پیشرفت قابل توجهی داشتیم و تعداد تهدید های وزیر ها به طور قابل توجهی کاهش یافتند.

مقایسه الگوریتم عقب گرد با الگوریتم ژنتیک و نتیجه گیری

در این مقایسه اومدیم با استفاده از تابع timeit پایتون حساب کردیم که این مقایسه را در جدول زیر ملاحظه می کنید:

تعداد وزیر	عقب گرد (ثانیه)	ژنتیک (ثانیه)
15	0.101	0.191
20	22.286	0.532
25	9.352	1.394
30	> 10 Minutes	4.077
35	-	5.127
40	-	8.890
45	-	19.229
50	-	43.670
55	-	52.941
60	-	85.146
80	-	291.414

روش اجرا پروژه

با cmd فولدر الگوریتم را باز میکنیم، سپس pip install -r requirements.txt را اجرا میکنیم تا پکیج های لازم را نصب کنیم.

برای اجرا فایل main.py با کد ادیتوری باز میکنیم و سپس کد زیر را در فانکشن main کپی میکنیم.

الگوریتم عقب گرد:

```
game = NQueenBackTracking(consts.N_COUNT, consts.BLOCK_SIZE)
game.set_up()
game.run()
game.finish()
```

الگوریتم ژنتیک:

```
        game = NQueenGeneticAlgorithm(consts.N_COUNT,
consts.BLOCK_SIZE)

            game.run()

            game.show_progress()

            game.set_up()

            game.finish()
```

منابع

https://towardsdatascience.com/genetic-algorithm-vs-backtracking-n-queen-problem-cdf38e15d73f

https://github.com/waqqasiq/n-queen-problem-using-genetic-algorithm

https://www.kancloud.cn/leavor/cplusplus/630541

https://en.wikipedia.org/wiki/Eight_queens_puzzle

https://fa.wikipedia.org/wiki/%D9%85%D8%B3%D8%A6%D9%84%D9%87_%DA%86%D9%86%D8%AF_%D9%88%D8%B2%DB%8C%D8%B1

https://aljrico.github.io/blog/genetic-algorithms