به نام خدا

پروژه ی دوم هوش مصنوعی:

در این پروژه قصد داریم بازی سودوکو همراه با رنگ ها را به کمک الگوریتم backtracking پیاده سازی بکنیم.

برای این کار از کلاس block که نشان دهنده ی هر بلوک از یک جدول n*n می باشد، کلاس board که نشان دهنده ی هر جدول n*n می باشد و کلاس Game که نشان دهنده ی مجموعه ی جدول ها و سیر حل مسیله است استفاده کردیم.

کلاس block :

هر بلوکی دارای یک ای دی می باشد که نشان دهنده ی سطر و ستون آن است که به صورت یک لیست دو تایی این ای را در نظر گرفته ایم. هر بلوک دارای رنگ و عدد می باشد و برای هر بلوک باید یک دامنه از رنگ ها و عدد ها اختصاص داده شود. برای هر بلوک میتوان مقدار هیورستیک درجه را محاسبه نمود و آن را به عنوان یک ویژگی بلوک درنظر گرفت. یکی از شرایطی که حین حل مسیله پیش می آید این است که رنگ و عددی که به هر بلوک در حین حل مسیله اختصاص دادیم با تناقض در مراحل بعدی مواجه شود و مجبور به عقبگرد شویم و مقدار اختصاص داده را عوض کنیم برای اینکه از عوض کردن رنگ ها و عدد هایی که در ابتوای مسپله به عنوان صورت سوال برای ما مشخص شده باشند اگر دو متغیر primary استفاده میکنیم که در صورتی که عدد یا رنگ ما جز داده ی مسیله باشند دچار تغییر نشوند. برای متغیر ها یک مقدار primaryDegree نیز در نظر میگیریم که نشان دهنده ی همه ی همسایه ها برای هر بلوک می باشد که در صورتی که در گوشه ها باشد عدد۲ و در حاشیه به جز گوشه ها باشد عدد ۳ و در غیر اینصورت عدد ۴ می باشد اما مقدار Degree همسایه های بدون عدد یا رنگ می باشد.(در ابتدای بازی ابتدا برای ساخت جدول فرض میکنیم جدول خالی و مقدار درجه همان تعداد همسایه های هر بلوک می باشد.)

کلاس Board :

برای مشخص کردن هر جدول نیاز است که دامنه ی رنگ و عدد و مجموعه ی بلوک ها را به آن اختصاص دهیم. برای ایجاد یک جدول در صورتی که مجموعه ی بلوک ها را نداشته باشیم میتوانیم از string ورودی که در صورت مسیله آمده استفاده کنیم و برای هر خانه از جدول یک بلوک در نظر بگیریم و مقدار آن را در صورتی که دارای عدد یا رنگ م باشد ست کنیم و مقدار primary رنگ و عدد را نیز برابر true قرار دهیم. در این حالت مقدار محدودیت ها را مقدار اولیه فرض میکنیم و به هر بلوک یک آی دی نسبت میدهیم. با اضافه کردن هر بلوک به blocksList یک جدول از بلوک های متعدد ساخته می شود. حال برای آپدیت کردن محدودیت ها از تابع updateConstraints استفاده میکنیم که در این تابع برای برای هر بلوک محدودیت ها را با توجه به قوانین و اعداد و رنگ های اولیه ست میکند.

این تابع بدین صورت است که ابتدا برای هر بلوک سطر و ستون آن را چک میکند و اعدادی که در سطر و ستون آن هستند را از مقادیر دامنه حذف میکند. سپس برای بلوک های مجاور هر بلوک، رنگ های آن ها را نیز از دامنه ی رنگ بلوک حذف میکند و درصورتی که بلوک های همسایه دارای رنگ و عدد باشد آن را به عنوان یک متغیر ست شده در نظر میگیرد، تعداد آن ها را می شمارد و از مقدار اولیه ی درجه کم کرده و به عنوان درجه برای بلوک ست میکند.

یکی دیگر از توابع مورد نیاز یافتن بلوکی است که دارای کمترین مقدار دامنه و کمترین مقدار رنگ باشد که این توابع مشابه هم نوشته شده اند. برای یافتن بلوکی که داری کمترین دامنه است ابتدا باید اندازه ی دامنه ی عددی تمام بلوک ها در یک جدول را مقایسه کنیم و کمترین مقدار دامنه را درنظر بگیریم. این مقایسه باید بین بلوک هایی باشد که دامنه ی آن ها صفر نیست و جز اعداد اول ورودی به برنامه نیستند و در ادامه برای مثال اگر کمترین مقدار دامنه ۱ باشد چندین بلوک این ویژگی را داشته باشند بنابراین نیاز هست که از هیوریستیک درجه برای انتخاب بین آن ها استفاده کنیم پس باید خروجی این تابع لیست باشد.

تابع دیگری که برای این کلاس تعریف کرده این تابع forwardChecking می باشد که برای این است که عد از آپدیت شدن هر مرحله توسط مقادیر اختصاص داده شده با این تابع چک کنیم آیا در ادامه به تناقض خواهیم خورد یا خیر. در صورتی که به تناقض میخوریم از ادامه ی حل جدول با آن مقادیر اجتناب کرده و مقادیر دیگری به متغیر نسبت میدهیم. برای چک کردن تناقض این مورد را بررسی میکنیم که ایا بعد از انجام اپدیت ایا دامنه ای از رنگ ها یا عدد ها تهی گردیده یا خیر که اگر تهی گردیده باشد باعث میشود به جواب درست جدول نرسیم سپس اگر خانه ی همسایه هر بلوک دارای رنگ و عدد بود باید اولویت رنگ را برای آن بررسی کنیم که اگر عدد کوچکتری دارد رنگی با اولویت کمتر داشته باشد.

کلاس Game :

در این کلاس دو تابع برای شروع و پایان بازی تعریف میکنیم. شروع بازی با string ورودی تعریف میشود و پایان بازی چک میکند که آیا همه ی خانه های جدول پر شده اند یا خیر.

تابع solve مسیله را به روش backtrack حل میکند بدین صورت که ابتدا چک میکند آیا جدول پر شده است یا خبر. اگر جدول پر شود به جواب مسیله رسیده ایم زیرا همه ی متغیرها بدون تناقض ست شده اند اما اگر جدول هنوز پر نشده باشد وارد حلقه while می شود. تمامی حالت ها و نود ها را که همان جدول Board می باشد درون یک لیست نگه داری میکنیم و در مرحله در صورتی که جدول کنونی دارای تناقض نباشد آن را به لیست جدول ها اضافه میکنیم. پس در هر مرحله اخرین جایی که باید شروع کنیم و روی آن تغییرات اعمال کنیم اخرین درایه از لیست جدول ها میباشد. درون این جدول متغیری را انتخاب میکنیم که از لحاظ تعداد دامنه کمترین مقدار را داشته باشد. اگر تمامی اعداد در جدول ست شده باشند سپس به سراغ انتخاب یک متغیر میرویم که رنگی نداشته باشد و از لحاظ دامنه ی رنگی هم کمتر از بقیه باشد. در هر مرحله اگر تعداد متغیر ها بیشتر از یک بود باید متغیری انتخاب شود که هیوریستیک درجه آن کمتر باشد.

سپس چک میکنیم که ایا متغیر نیاز به عدد و رنگ یا رنگ یا عدد دارد. با توجه به اینکه به کدام نیاز دارد یک مقدار را ددر دامنه ی متغیر در نظر میگیریم و تغییرات رنگ و عدد را اعمال میکنیم و محدودیت های جدول را آپیدیت می کنیم حال چک میکنیم که ایا جدول دچار تناقض نباشد اگر دچار تناقض بود باید یک مقدار دیگر ست کنیم و اگر دچار تناقض نبود جدول جدید را به لیست جدول ها اضافه میکنیم و دوباره تابع solve را صدا میزنیم.

این تابع آن قدر ادامه پیدا میکند که بتواند به جدول کاملی برسد و در هر مرحله با مشاهده ی تناقض یک مرحله به عقب باز میگردد یعنی رفتاری مشابه DFS دارد.