# گزارش پروژه ی میانترم

در این پروژه قصد داریم مسیله ی 8puzzle را به کمک الگوریتم های مختلف سرچ حل کنیم.

ابتدا کلاس Node را تعریف میکنیم. به کمک این کلاس هر نود به همراه ویژگی های آن تعرفی می شود.

کلاس Node:

این کلاس مشخص کننده ی ویژگی های هر نود می باشد. در این کلاس اعداد پازل در قالب یک بردار یک بعدی ۹ تایی نشان داده شده اند.(به جای بردار دو بعدی و برای راحتی محاسبات آن را به صورت یک بعدی در نظر گرفته ایم.)

هر نود دارای یک نود پدر می باشد. این نود را با استفاده از smart pointer تعریف کرده ایم. علاوه بر پدر هر نود دارای لیستی از فرزندان نیز می باشد. می توانستیم تایپ متغیر های والد و فرزندان را از نوع Node قرار دهیم ولی چون ممکن است نود های زیادی تولید شود و همه ی آن ها باید در حافظه نگه داری شود بنابراین نگه داری آن ها به صورت پوینتر مناسب تر می باشد. برای این امر از اسمارت پوینتر ها استفاده میکنیم و دلیل آن این است که حالتی وجود دارد که چندین نود دارای یک والد می باشد پس چندین پوینتر دارای رفرنس یکسانی می شود بنابراین نمیتوان از پوینتر های ساده استفاده نمود.

جای خالی در پازل را با عدد صفر نشان می دهیم و اندیس آن را ذخیره میکنیم.(البته این اندیس در ابتدا صفر می باشد و در ادامه در توابع مشخص میگردد)

برای الگوریتم *A نیاز به تعریف تابع هیوریستیک و محاسبه ی آن برای هر نود می باشد باید تابع هیورستیک به درستی تعریف شود و این تابع باید سازگار باشد به صورتی که هزینه خروجی این تابع از هزینه ی واقعی کمتر باشد در این صورت مطمین خواهیم بود استفاده ازین تابع بهترین مسیر را به ما میدهد. برای این مسئله تعداد خانه هایی از پازل که در جای خود نیستند را درنظر میگیریم. (هیوریستیک های دیگری هم برای این پازل امتحان کردم نظیر مجموع فاصله ی هر عدد تا نقطه ی مطلوب، تعداد اعداد نابرابر در سطر و ستون و ... ولی در عمق بیشتری جواب را می یافتند.)

در هر مرحله برای انتخاب یک نود باید بررسی شود که ایا آن نود در مجموعه frontier (مجموعه ی نود های پیش رو در صف برای بسط داده شدن) و مجموعه ی explored (مجموعه نود ها یبسط داده شده) نباشد. برای راحتی این مقایسه در هر مرحله به هر نود یک id که عدد نه رقمی با اعداد پازل هستند را نسبت می دهیم در این صورت مقایسه با سرعت بیشتری انجام خواهد شد. در تابع تست هم این مقایسه با آی دی نود هدف انجام میپذیرد.

برای تولید فرزندان هر نود نیاز است که اول مکان صفر را بیابیم زیرا حرکات مجاز بر اساس مکان عدد صفر مشخص می شود. برای این کار از کتابخانه ی STL و تابع find استفاده کردیم که در صورتیکه ۰ را یافت کند برای یافتن اندیس آن بایدiterator را از iterator که به اول لیست اشاره میکند کم کنیم. پس از سافتن مکان صفر حال باید حرکات مجاز آن را تشخیص دهیم و سپس نود فرزند را برای هر کدام تشکیل داده و لیست فرزندان را آپدیت کنیم. مثلا برای حرکت راست زمانی که عدد در ستون سمت راست باشد نمیتواند به سمت راست حرکت کند پس باید باقی مانده ی اندیس عدد به ۳ کمتر از ۲ باشد. برای حرکت به سمت چپ نباید عدد در ستون سمت چپ باشد بنابراین باقی مانده ی اندیس آن به ۳ باید بیشتر از ۰ باشد. برای حرکت به سمت بالا عدد نباید در سطر اول باشد یعنی حاصل تفریق اندیس عدد از ۳ باید از ۰ بیشتر باشد و برای خرکت به سمت پایین عدد نباید در سطر آخر باشد بنابراین حاصل جمع اندیس عدد و ۳ نباید از ۹ بزرگتر باشد.

برای نمایش هر نود از رنگ ها استفاده کردیم که هر رنگ دارای کد مشخص می باشد. هر رنگ برای رنگ نوشته و رنگ پس زمینه دارای کد های مختلفی است و برخی ویژگی ها مانند bold,italic,underline,... نیاز در صورت نیاز دارای کد مخصوص می باشند. برای استفاده از رنگ ها همه ی کد های آن ها در فایل color.h قرار داده ایم. برای هر رنگ باید ابتدا کد esc + کد رنگ + m(جهت خاتمه) را قرار دهیم. پس از تغییر می توانیم از کد RESET استفاده کنیم تا تنظیمات به حالت اولیه برگردد.

چندین operator مساوی تعریف گشته است که در ادامه نیاز به مقایسه ی objectها و پوینتر های objectها بوده و بر اساس نیاز operator ها تعریف گشته است.

یکی از کارهایی که برای راحتی کار انجام میدهیم بررسی حل شدنی یا نشدنی بودن پازل قبل از بررسی آن با الگوریتم ها می باشد. این روش ذکر شده و ئیاده سازی شده برای حالت پایانی {1,2,3,4,5,6,7,8,0} صادق است و برای بقیه ی حالت ها باید با الگوریتم ها بررسی گردد که ایا جواب دارد یا خیر.  به همین دلیل است که برای تابع isSolvable درون الگوریتم ها شرط هدف مساوی با مقدار ذکر شده لحاظ میگردد.(در ابتدا برای تولید پازل رندم هرف را همان دیفالت درنظر میگیرم و پازلی تولید میکنیم که با این شرط قابل حل باشد.) در این روش زوج هایی را می یابیم که عدد بزرگتر در جایگاه کوچکتر باشد. برای مثال زوج (6,7) درست و زوج (7,6) نادرست است. حال اگر تعداد این زوج های نادرست زوج باشد پازل حل شدنی و اگر فرد باشد پازل حل نشدنی است.

کلاس BFS:

این کلاس شامل نود ریشه، لیست frontier می باشد و برای جست و جوی در نود های بسط داده شده یا پیش رو از دو لیست nfrontier , nexplored استفاده می کنیم و در آن ها ای دی منحصر به فرد هر نود را قرار میدهیم.(آی دی هر نود عدد ۹ رقمی حاصل از پازل آن می باشد) نیاز به لیست frontier داریم زیرا باید در هر مرحله یکی از نود های آن را انتخاب و بسط دهیم پس این لیست را به صورت جداگانه از پوینتر هایی به نود ها تشکیل میدهیم. در این کلاس دو لیست path,actions را برای برگرداندن راه حل استفاده میکنیم.

تابع constructor این کلاس با نود ریشه ساخته می شود و دادن پازل خروجی اختیاری است و مقدار دیفالت برای آن در نظر گرفته شده است.

تابع contains یک لیست که مجموعه ای از آی دی های نود ها می باشد به همراه یک آی دی دریافت میکند و وجود ان را در لیست بررسی میکند.

الگوریتم BFS بدین صورت است که ابتدا باید نود ریشه را چک کنیم که آیا هدف می باشد یا خیر. در صورتی که هدف باشد باید آن را برگردانیم و در غیر این صورت بررسی میکنیم آیا پازل با این حالت اولیه قابل حل می باشد یا خیر و در صورت قابل حل بودن این نود باید به مجموعه ی frontier و آی دی آن به nfrontier اضافه گردد. حال وارد حلقه while می شویم.این حلقه تا زمانی ادامه پیدا میکند که یا هدف پیدا شود یا مجموعه ی frontier تهی گردد. در این حلقه ابتدا اولین نود frontier را برداشته و آی دی نود آن را به explored اضافه میکنیم و سپس از مجموعه ی frontier و nfrontier حذف میکنیم. پس در هر مرحله یک سری نود به انتهای frontier اضافه میشود و در اول حلقه اولین نود جهت بررسی برداشته می شود. با توجه به حالتی که برای frontier پیش می اید باید آن را به صورت صف FIFO پیاده سازی کنیم. برای این کار میتوانیم از queue استفاده کنیم یا میتوانیم ز deque استفاده کنیم. با وجه به اینکه deque سرعت کمتری از queue دارد پس از گزینه ها حذف میگردد.( اولین پیاده سازی الگوریتم با استفاده از deque بوده زیرا قابلیت و حذف و اضافه از هر دو سمت را دارد ولی سرعت بسیار پایین برنامه شد پس آن را جایگزین کردیم.) queue قابلیت صف FIFO را به خوبی دارد اما در این الگوریتم بیس

کار بر اساس مقایسه می باشد. یعنی در طول برنامه تعداد مقایسه بسیار زیاد است که در این امر queue نسبت به vector سرعت کمتری دارد. بنابراین از vector استفاده میکنیم. علی رغم اینکه ماهیت وکتور برای حذف از ابتدا نمی باشد اما سرعت iteration در ان برای مقایسه بالاتر بوده لذا این container را انتخاب میکنیم. حال نود جهت بسط دادن را انتخاب کردیم. باید در این مرحله فرزندان آن را به دست بیاوریم و هر فرزند را قبل از اضافه کردن به مجموعه ی frontier باید چک کنیم که نود تکراری نباشد.(با این کار از ایجاد لوپ در درخت جلوگیری میکنیم.) در این مرحله دوباره به ابتدای حلقه بازمیگردیم.

همانگونه که مشخص شد الگوریتم BFS الگوریتمی اول سطح است زیرا ابتدا همه ی نود های یک سطح را بسط می دهد و سپس به سراغ سطر بعدی میرود. در این الگوریتم هدف در کمترین سطح ممکن یافت میشود.( سطر ها و عمق ها را از ۰ در نظر گرفته ایم)

در تابع pathtrace از نود هدف شروع کرده و با توجه به نود های والد مسیر رسیدن به حالت اولیه را دنبال میکنیم و آن را درون لیست path قرار میدهیم و در انتها لیست را معکوس میکنیم تا مراحل از ابتدا به انتها نمایش داده شوند. برای محاسبه ی عمق نیز چون عمق از صفر شروع شده است باید سایز path منهای یک گردد.

<u>الگوریتم DFS:</u>

این الگوریتم به اول عمق معروف است. بدین صورت که ابتدا بک نود تا تا عمقی که میتواند بسط می دهد و سپس به سراغ نود های دیگر میرود. اگر هدف در عمق زیاد باشد این الگوریتم میتواند بهینه تر از اول سطح عمل کند اما به طور کلی ازین الگوریتم به تنهایی استفاده نمیشود بلکه برای آن حد تعیین میکنند و الگوریتم عمقی محدود شده را DLS می نامند به طوری که الگوریتم اول عمق را تا حد تعیین شده اجرا میکند و نه بیشتر و اگر تا آن حد جواب یافت نشود حالت cut-off رخ میدهد.

این کلاس دقیقا مشابه کلاس BFS می باشد. در این کلاس همانند قبل مقدار دیفالت نود هدف را در ارایه goal ذخیره کرده ایم و هنگام چک کردن حل پذیر بودن پازل این شرط نیز باید اعمال شود.

در این کلاس لیستی از نود های explored نیاز است علاوه بر کلاس اول سطح زیرا باید در مقایسه ها نودی که قبلا بسط داده شده بوده اما در سطح پایینتر بوده مانع بسط نودی با همان مقدار پازل در سطح بالاتر فرزندان دیگر نشود زیرا ممکن است با بسط آن نود به هدف برسیم.

همین منطق برای frontier صادق است. نودی که در frontier قرار دارد اما در سطح بالاتر نباید مانع بسط نود انتخاب شده برای بسط در سطح پایینتر گردد.

برای این کلاس یک پارامتر دیگر به نام depth برای مقایسه ی عمق نود ها اضافه میشود. عمق هر نود هنگام تولید تعیین می شود و برابر با عمق والد به اضافه ی ۱ است.

برای این الگوزیتم ابتدا ریشه را به مجموعه ی Frontier و آی دی آن را به nfrontier اضافه میکنیم. سپس تابع بازگشتی را صدا میزنیم.

تابع بازگشتی یک پوینتر به نود و حد را به عنوان ورودی قبول میکند. در این تابع ابتدا چک میکند ایا نود ورودی هدف می باشد یا خیر. سپس قابل حل بودن آن را بررسی میکند. سپس بررسی میکند که آیا حد آن از ۰ کمتر مساوی شده یا خیر زیرا اگر limit=0 شود حالت cut-off اتفاق افتاده و نود باید از مجموعه ی frontier حذف و به explored اضافه شود. در غیر اینصورت نود همان frontier.back() می باشد که باید بسط داده شود زیرا صف در این الگوریتم LIFO می باشد. همانند قبل

نود را بسط می دهیم و اگر هدف نبود و در مجموعه های frontier و explored با شرایط گفته شده وجود نداشت به لیست اضافه میگردد.

سپس پس از یافتن تمامی فرزندان نود و اضافه کردن به frontier باید مجموعه ی فرزندان را reverse کرده و سپس تابع ازگشتی را روی هز یک از فرزندان با حد یکی کمتر ضدا بزنیم. دلیل reverse کردن آن است که ()frontier.back در هر مرحله همان نود ورودی است بنابراین اخرین فرزند وارد شده می باشد پس برای این برابری نیاز به معکوس کردن داریم و یکی یکی از اول تابع بازگشتی را صدا میزنیم.

الگوریتم DLS برای زمانی که ندانیم جواب در چه عمقی است ممکن است آزار دهنده باشد زیرا باید با ازمون و خطا عمق جواب را پیدا کنیم. برای حل این مشکل از الگوریتم IDS استفاده میکنیم. در این الگوریتم از عمق صفر تا یک عدد بزرگ برای مثال طول stack و غیره (در کد ۵۰۰۰ درنظر گرفته شده است با توجه به اینکه طول stack حدود ۷۰۰۰ می باشد بنابراین عدد معقولی است ۵۰۰۰) هر بار DLS را صدا میزنیم. بنابراین هدف در کمترین عمق ممکن یافت میشود.(البته این الگوریتم زمانبر است زیرا برای هر عمق DLS را صدا میزند ولی هدف را در عمق کمتر پیدا میکند)

الگوریتم *A:

این الگوریتم بر اساس تابع هیوریستیک که تعریف کردیم به هر نود یک مقدار اختصاص می دهد و در هر مرحله نودی را انتخاب میکند که هیوریستیک کمتری داشته باشد. این روش نسبت به دو روش قبلی سریعتر می باشد و تعداد نود کمتری را برای رسیدن به هدف بسط می دهد. البته باید هیوریستیک مناسب و سازگاری انتخاب کنیم تا جواب بهینه به دست بیاید.

در این الگوریتم علاوه بر تابع هیوریستیک h(n) نیاز به یک تابع هزینه برای action ها نیاز داریم که در این مسئله هزینه ی هر اکشن را برابر ۱ در نظر میگیریم و به طور کلی g(n) برابر با عمق هر نود می شود.

$$F(n) = g(n) + h(n)$$

در این الگوریتم ابتدا هدف، حل پذیر بودن مسئله چک می شود و سپس نودی را انتخاب میکنیم که دارای کمترین مقدار هیوریستیک(در اینجا همان f(n) ) باشد. سپس آن نود را بسط می دهیم و فرزندان آن را پیدا میکنیم و در صورت تکراری نبودن آن را به مجموعه frontier اضافه میکنیم. در صورتی که نود جدید در explored وجود داشته باشد، اگر مقدار هیوریستیک آن کمتر است باید نود را از explored خارج کرده و به frontier اضافه کنیم. و اگر نود جدید در frontier موجود باشد و مقدار هیوریستیک آن کمتر باشد باید آن نود را از مجموعه ی frontier خارج کرده و نود جدید را جایگزین کنیم.

قسمت UI:

در این قسمت برای دو حالت برای ورودی در نظر گرفتیم یکی حالتی که به تعداد مورد نظر نود اولیه را shuffle کند و دیگری آن که ورودی را به صورت عددی وارد کند. برای حالت خروجی نیز دو حالت در نظر گرفتیم یکی حالتی که خروجی را وارد کنیم و دیگری حالتی که از خروجی دیفالت مسئله استفاده کنیم.

گزینه ها را در این حالت به صورت حروف الفبا درنظر گرفتیم بنابراین برای دریافت گزینه ها از getchar استفاده میکنیم. سپس تا زمانی که گزینه ی مد نظر را کاربر انتخاب نکرده منتظر می مانیم تا دوباره تلاش کند و گزینه ی مناسب را انتخاب کند. سپس بر اساس حالت های مختلف شرط هایی در نظر گرفتیم. برای دریافت تعداد shuffle که یک int است از cin استفاده کرده و برای دریافت پازل از getline استفاده میکنیم. به صورتی که اعداد پازل با فاصله باید وارد شوند و درانتها * زده شود تا درفات تمام شود.

سپس در صفحه ی بعد الگوریتم ها را مشخص کرده که میتوانیم انتخاب کنیم. در این صفحه برای برگشت به منوی اصلی می توان q را فشار داد.

این دو تابع را درون یک while اجرا میکنیم و با while((ch=getchar())!= 'q') منتظر می مانیم تا کاربر q را فشار دهد و برنامه به منوی اصلی بازگردد.

لینک repository :

https://github.com/z-arabi/8puzzle.git

* این repository دارای یک شاخه BFS می باشد. در شاخه ی اصلی برنامه همراه با ارث بری نوشته شده بود که سرعت پایینی داشت بنابراین یک شاخه ی جدا برای بدون ارث بری درست شد که نتیجه ی مطلوب هم داد.(این دو را با هم merge نکردم زیرا امکان conflict وجود دارد و باید آن ها را رفع کنم) پروژه ی نهایی در branch BST قرار دارد.

متشکرم از شما