پروژه میان ترم

اول از همه بعد از ایجاد ریپازیتوری در گیت هاب میک فایل و داکر فایل را ایجاد کردم . بعد از دیدن مقدار زیادی ویدیو راجع به این نوع پازل متوجه شدم که می توان آن را از ۲ راه حل کرد راه اول استفاده از گره است و راه دوم استفاده از حالت هایی است که پازل می تواند داشته باشد، من راه دوم را انتخاب کردم.

Base of class

اول کلاس State را ساختم ، در این پازل ما ۴ نوع حرکت داریم ، بالا ، پایین ، چپ و راست و به ترتیب توابع آن ها را باید ایجاد می کردم هم چنین تابعی لازم است که مقدار خانه ی خالی و کناری آن را تعویض کند،

یک default constructor گذاشتم و یک constructor که یک پازل که جایگاه خانه ی خالی را به عنوان ورودی می گیرد و در یک پازل کلی ذخیره می کند مختصات خانه ی خالی را نیز ذخیره می کند.

یک بردار ایجاد کردم که مسیر راه حل را در آن ذخیره کنم . از آنجایی که حل پازل با الگوریتم های دیگر است تابع ای حرکت از جنس bool هستند.

برای تابع movefreeup باید اول بررسی کنیم که خانه ی خالی کجاست پس ۲ تابع getfreeX / getfreeY را نوشتم که مختصات آن را برمی گردانند.

حال برای movefreeup با توجه به این ردیف اول ۰ است تنها زمانی می توانیم حرکتی داشته باشیم که در ردیفی غیر از ردیف اول باشیم پس getfreeX()>0 حالا باید مقدار ۰ را با خانه ی بالایی عوض کنیم پس تابع swapTileValue را نوشتم که حالت ورودی و مختصات خانه ی غیر خالی را می گیرد و برابر مختصات خانه ی ۰ قرار می دهد و بعد عدد خانه ی مختصات ورودی را صفر می کند.

در تابع movefreeup مختصات Y ثابت است و X تغییر می کند پس با استفاده از swapTileValue مقدار ۰ را با خانه ی بالایی آن عوض می کنم حالا لازم است که مختصات خانه ی ۰ آپدیت شود ، تصمیم گرفتم برای این کار یک تابع بنویسم به اسم setFree که همین کار را انجام می دهد. و بعد حرف U را در وکتور path می ریزم.

۳ تابع حرکت دیگر نیز به همین شکل پیاده سازی شدند.

BFS

اول سعی کردم همان طور که در geeksforgeeks توضیح داده شده بود آن را پیاده سازی کنم . همچنین اپاتور های == و = را نیز پیاده سازی کردم تا بتوانم ۲ پازل را به هم خانه به خانه مقایسه کنم و یا پازلی را در دیگری بریزم.

روش geeksforgeeks با استفاده از node بودش و آنچه من می خواستم را انجام نمی داد و تصمیم گرفتم که با استفاده از template ها آن را پیاده سازی کنم. (templateها می توانند سریع تر از توابع کلاس باشند)

با توجه به این که ای الگوریتم با اسفاده از queue هستش پس حتما به عنوان ورودی باید یک queue بگیرد که در آن پازلی که می خواهیم حل کنیم وجود دارد. همچنین باید حالت نهایی پازل را به عنوان یک objectاز کلاس نیز بگیرد(این شرط را برای این گداشتم که بعدا بتوانم برای حالت دلخواه نیز حل کنم) همچنین از یک set برای نگه داشتن حالت های بررسی شده استفاده کردم و همچنین یک object از کلاس به عنوان solution نیز می گیریم.

در BFS ما تا زمانی که queue خالی نشده هست خانه ی اول را که یک حالت از تمام حالت هایی که پازل می تواند باشد را در در یک حالت از جنس کلاس دخیره می کنم و بعد آن حالت را از queue حذف می کنم حالا اگر این حالت با مقدار goal برابر باشد پس جست و جو ما تمام شده است و برابر solution قرار می دهم وگرنه باید بیایم برای حالت اولیه تمام حالت هایی که با حرکت خانه ی خالی ایجاد می شود را بدست آورم و در یک وکتور بریزم همچنین حالت اولیه که s بود را به عنوان حالت بررسی شده درون set (به اسم closed) می ریزم که دیگر بررسی نشود. سپس باید تمام حالت های موجود از حرکت خانه ی خالی را درون queue بریزم و دوباره این روند تکرار شود تا به حالت دلخواه برسم.

expand

برای بررسی تمام حالت های موجود از حرکت خانه ی خالی در کلاسم تابع expand را نوشتم که یک object به اسم child می سازد و بعد ۴ حرکت را روی آن بررسی می کند و اگر قابل انجام شدن باشند child جدید را درون وکتوری از جنس کلاس می ریزد و در آخر آن را برمی گرداند.

findFreex – findFreeY-toString

برای پیدا کردن مختصات صفر پازل ورودی ۲ تابع findfreex و findfreey را نوشتم و برای نمایش پازل ها تابع toString تا با شکلی زیبا نمایش داده بشود.

getpath

همچنین برای نماش راه حل از تابع getpath استفاده کردم که اعضای وکتور path را به صورت رشته در می آورد و نمایش می دهد.

operator<

همچنین به دلیل استفاده از set نیاز به یک اپراتور < بود .

noOfMoves

برای بدست آوردن تعداد حرکات با تابع noOfMoves سایز وکتور path را بر می گردانم.

show

برای نمایش هر مرحله از راه حل با استفاده از تابع show برای پازل اولیه اعضای وکتور path را بررسی می کنیم و با توجه به آن حرکت را روی object child از کلاس اعمال می کنیم و چاپ می کنیم.

DFS

برای dfs نیز مانند bfs عمل کردم با این تفاوت که با stack باید می نوشتم و LIFO بهستش که البته dfs می تواند گم بشود برای همین علاوه بر خالی نبودن stack برای آن سایز نیز تعیین کردم بعد از چند بار ران با عدد ۷۰۰۰ برنامه killed نمی شد.

Get_user_input

این تابع پازل ورودی را خانه به خانه از کاربر دریافت می کند در یک وکتور می ریزد و در startingboard ذخیره می کند.

Get_user_input_goal

مانند Get_user_input اگر کاربر بخواهد تابع Get_user_input را صدا می کند تا پازل را کاربر وارد کند وگرنه همان حالت دیفالت را می دهد.

Random_Board

این تابع ۹ بار عددی رندوم بین ۰ تا ۹ تولید می کند و در یک unordered_set می ریزد تا تکراری نباشد و ترتیب آن نیز بهم نریزد و سپس مقادیر را در یک وکتور می ریزد سپس هر عضو وکتور را به خانه ای از پازل اختصاص می دهد.

getInvCount-isSolvable

برای اینکه ببینیم آیا یک پازل قابل حل است همان طور که در geeksforgeeks نوشته بود باید بررسی کنیم که تعداد زوج های وارونه ی آن زوج است یا نه تنها در این حالت پازل قابل حل است .

*زوج وارونه مثل (۷و۸)

Menu

برای بهتر شدن ظاهر یک سری ansi کد پیدا کردم همچنین این قابلیت را دارد که در dfs مراحل را نبینیم و فقط راه حل را ببینیم و می توان دستی عمق تعریف کرد وگرنه دیفالت دارد و به زیبایی همه چیز را نمایش می دهد.

تمامی int ها را به size_t تبدیل کردم تا در کامپیوتر ۳۲ نیز قابل اجرا باشد همچنین استفاده از refrence ها در توابع از کپی اضافه جلوگیری می کند و باعث افزایش سرعت می شود.

Git hub

https://github.com/nigkty123/Ap1399-1-midproject.git