پارسا محمدپور – 98243050

تابع bfs:

در این تابع با توجه به نوع الگوریتم bfs (که در آن ابتدا node های با عمق یکسان را بسط می دهیم) جلو رفتیم.
ابتدا تابع percpet را صدا کردیم تا موقعیت اولیه و صفحه را ست کند.
سپس یک صف (queue) از نوع لیست می باشد ولی با آن برخوردی مطابق با یک صف داشتیم) ایجاد کردیم و در آن node هایی که آن ها را بررسی می کنیم را قرار می دهیم(در واقع open list می باشد). سپس node عه start را در آن قرار دادیم تا بتوانیم کار را شروع کنیم.
یک متغییر به اسم find داریم که در آن یک مقدار Boolean قرار می گیرد که نشان دهنده این است که آیا ما در در ماتریس می توانیم به جوابی برسیم و راهی پیدا کنیم یا خی. مقدار اولیه آن را برابر با False قرار دادیم و در صورتی که در اجرای الگوریتم، node عه goal را پیدا کنیم، مقدار آن را برابر با True قرار می دهیم و سپس با روش گفته شده در ادامه، مسیر را رنگی می کنیم.
یک دیکشنری (passed) تعریف می کنیم که در اصل این دیکشنری برای این است که هر وقت توانستیم به جواب برسیم، بتواین مسیری که با آن به جواب رسیدیم را پیدا کنیم. محوه کارکرد کلی این دیکشنری به این صورت است که هر node ای را که می خواهیم به لیست باز مان (متغییر queue) اضافه کنیم، آن را node عه child را به عنوان key قرار می دهیم و آن node عه parent را به عنوان value عه آن node عه child قرار می دهیم. سپس هر سری که می خواستیم یک node عه جدید را به لست باز مان اضافه کنیم، چک می کنیم که آیا این node در key های این دیکشنری قرار دارد یا خیر، اگر قرار داشت یعنی این node قبلا به لیست بازمان اضافه شده است و نیازی به اضافه کردن دوباره آن نیست.(برای شروع هم چون node عه start اولین node ای است که بررس می شود، آن را به دیکشنری مان اضافه می کنیم(البته چون از جایی به آن نرسیدیم، صرفا مقدار value آن را خودش می گذاریم) تا دیگر وقتی که بچه هایش را بررسی می کنیم آن را دوباره به لیست باز مان اضافه نکنیم.)
درون یک حلقه تا وقتی که لیست بازمان خالی نشده باشد پیش می رویم (یعنی node ای برای بسط کردن وجود داشته باشد).
در هر مرحله اجرای این حلقه اولین عضو این لیست باز (برای این اولین عنصر را در نظر میگیریم چون که اولین عنصر در پایین ترین عمق می باشد و بدین صورت این لیست بازمان، عملکردی مشابه با یک صف خواهد داشت) را می گیریم و از صف آن را پاک می کنیم. اگر این عنصر node عه goal بود، که مقدار find را برابر با True می کنیم و از حلقه خارج می شویم، در غیر این صورت، تمامی بچه های(همسایه های) این node را در صورتی که block نباشند و آن ها را قبلا بررسی نکرده باشیم (در key های دیکشنری ما (متغییر passed) وجود نداشته باشند) و همچنین x و y آن node ، موجود و قابل قبول باشد، آن را به لست اضافه می کنیم و در دیکشنری هم با توجه به روال گفته شده آن را قرار می دهیم.
ترتیب بررسی و اضافه کردن node های همسایه را به گونه ای قرار داده ایم که اولویت با node عه بالایی، سپس سمت راستی، سپس پایینی و در آخر هم سمت چپی باشد.(ترتیب اضافه شدن آنها این گونه می باشد که چون هر سری اولین عنصر لیست باز را میگیریم، پس اولویت به این شکل در می آید.)
همچنین در این الگوریتم، آزمون هدف در لحظه ایجاد است، پس هر گاه که یک node عه child را می خواهیم اضافه کنیم به لیست بازمان، چک می کنیم که آیا آن node ، node عه هدف است یا خیر.

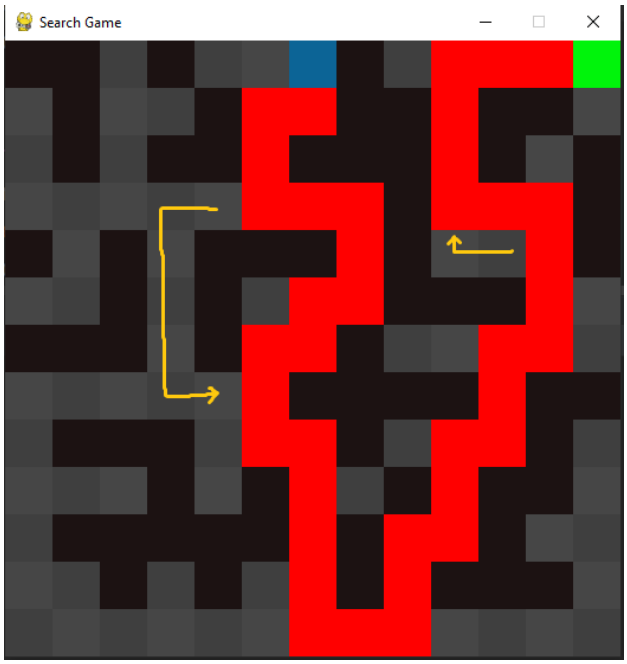
در انتها برای رنگ کردن node های موجود در مسیر پیدا شده، (همانطور که گفته شد، اگر مسیری پیدا شده باشد) در یک while پیش می رویم تا وقتی که به node عه start نرسیده باشیم، همچنین قبل از اجرای حلقه، به parent عه node ای می رویم که نود نهایی(node عه goal که اگر به جواب رسیده باشیم، همین node می باشد) را به عنوان جواب داده بود.
در هر سری اجرای حلقه، آن node را برابر با node عه parent ش قرار می دهیم تا همینطوری، node هایی که این مسیر را ساخته اند را پیدا کنیم.
اردر زمانی ما b^d می باشد. از جایی که می توان **متوسط** تعداد node های اضافه شده در هر حالت را 2 تا در نظر گرفت، پس اردر ما نمایی می باشد.

شکل اجرای الگوریتم bfs :

با توجه به اینکه مسیر پیدا شده در این شکل، کمترین میزان هزینه را دارد، پس این الگوریتم درست کار می کند.

اگر در شکل زیر، مسیر های زرد رنگ هم انتخاب می شدند، باز هم هزینه مسیر بهینه بود، اما همانطور که توضیح داده شد، به خاطر اولویت ها مسیر های زرد رنگ انتخاب نشدند.

(مسیر زرد رنگ سمت چپی به خاطر اولویت بیشتر بسط دادن سمت راست، به بسط دادن سمت چپ انتخاب نشد و مسیر زرد رنگ سمت راست هم به خاطر اولویت بالا تر جهت بالا به جهت چپ انتخاب نشد)



تابع dfs:

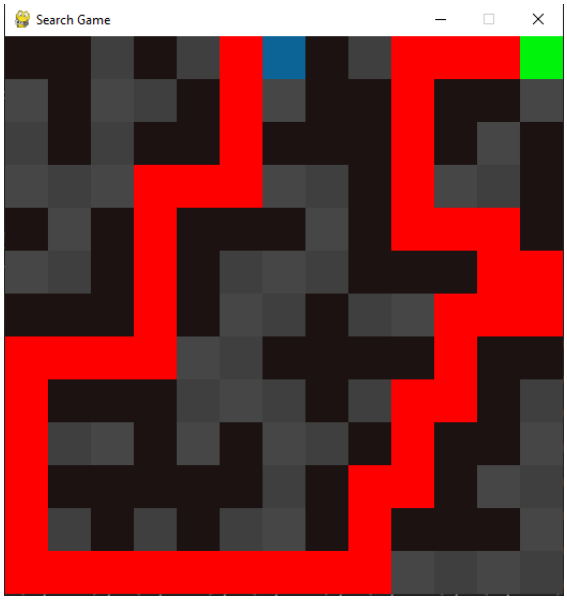
در این تابع تا حد خیلی خیلی زیادی مشابه با تابع bfs جلو می رویم. تنها دو جا با تابع bfs فرق داریم. یکی در جایی که می خواهیم در حلقه node ای را که بسط می دهیم از لیست باز انتخاب کنیم و دیگری در اضافه کردن node ها.
در این تابع باید از stack استفاده می کردیم. برای همین متغییر stack را تعریف کردیم که در اصل یک نوع list است، اما از آن طوری استفاده می کنیم که کارکردی مشابه با یک stack داشته باشد. هرسری برای آن که node ای را بسط دهیم، آخرین عنصر موجود در این لیست را pop می کنیم. به این شکل، این متغییر ، کارکردی مشابه با یک stack خواهد داشت.

اما دومین فرقی که با تابع bfs دارد در جایی است که می خواهیم فرزندان یک node ها را لیست بازمان اضافه کنیم. در تابع bfs چک می کردیم که آیا node عه child در key های دیکشنری می باشد یا خیر، اما اینجا چک می کنیم که در value های این دیکشنری است یا خیر. اما چرا ؟

در هنگامی که ما در الگوریتم bfs می آمدیم و در قسمت key ها چک می کردیم که آیا در این قسمت می باشد یا خیر. این باعث می شد که اگر node ای قبلا بسط داده شده است، دیگر به سراغش نرویم و کاری به آن نداشته باشیم. در قسمت key ها node هایی قرار داشتند که به gdsj fhc اصافه می شدند و وقتی که برای اولین بار به لیست اضافه شده باشند، یعنی بهترین مسیر ممکن به آن node از طریق همان node ای است که بدست آمده است چون مینیمم فاصله را داشته. (ما ابتدا node هایی را اضافه می کنیم که فاضله شان کمترین مقدار ممکن باشد) پس یعنی ما بهترین حالت رسیدن به آن node را در اختیار داشیم و دیگر نیازی به اضافه کردن آن node از مسیر های دیگر (که احتمالا طولانی تر هم بودند و یا در بهترین حالت هزینه کسان داشتند) برای آن node نبود.
اما در الگوریتم dfs مینیمم فاصله یا همان فاصله بهینه مهم نیست، بلکه ما باید صرفا ابتدا (با توجه به اولویتمان) یک سری node ها را جلو برویم و با توجه به آن حرکت کنیم. تنها نباید node های تکراری ای را که قبلا در لیست باز آن ها را بررسی کردیم دوباره ببینیم. حالا این نود هایی که در لیست باز انتخاب شدند و بسط داده شند دقیقا در قسمت value های این دیکشنری می باشند.(در الگوریتم dfs اگر یک node در لیست باز قرار گرفت، اما به خاطر node های قبلی اش، دوباره در لیست باز قرار گرفت، باید سری دومش را لحاظ کنیم و آن را بسط دهیم به خاطر اینکه ساختمان داده ما stack است و اینکه باید در عمق حرکت کنیم.)
(هر node ای که در لیست باز باشد، در قسمت key های دیکشنری هم هست، ولی هر node ای که بسط داده شده باشد، در قسمت value های دیکشنری می باشد.)

همچنین در این تابع، دیگر قبل از شروع حلقه کلی، (با توجه به تغییر گفته شده در بالا) نیازی نیست که node عه start را در دیکشنری اضافه کنیم.

شکل رنگ آمیزی شده با این الگوریتم به صورت زیر است:



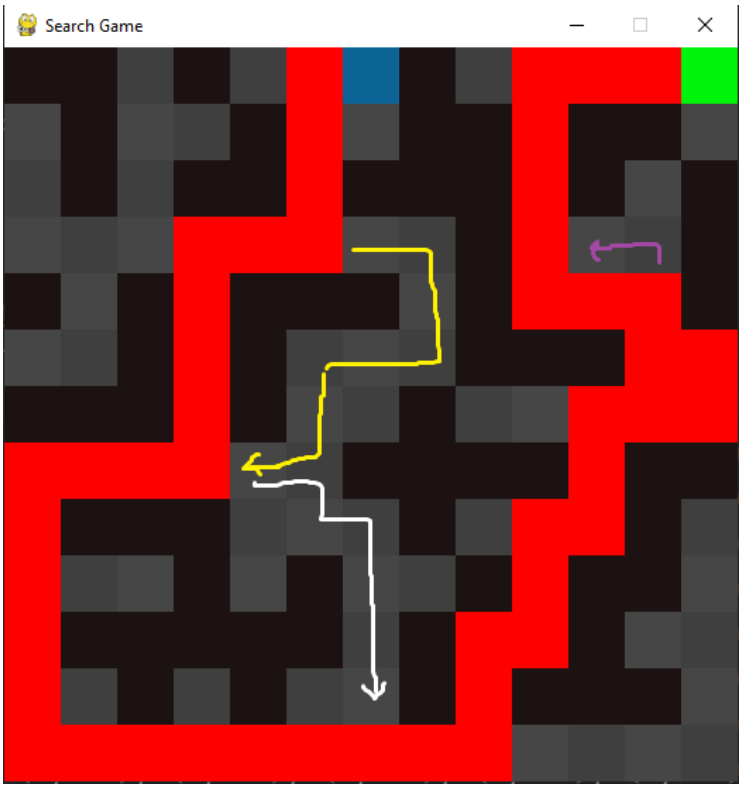
در اجرای الگوریتم dfsبه لیست بازمان، ابتدا node های بالایی، سپس سمت راستی، سپس پایینی و در نهایت node عه سمت چپی را اضافه کردیم. البته چون ساختمان داده مان stack است، پس طبیعتا آخرین عنصر اضافه شده، زود تر از بقیه بسط داده می شود، پس بنابراین اینجا اولویت هایمان دقیقا برعکس تابع bfs خواهد بود. یعنی node عه سمت چپی هر سری بالاترین اولویت را دارد و زودتر بسط داده می شود، سپس Node عه پایینی و سپس راستی و در انتها هم node عه بالایی.

دقیقا به خاطر همین اولویت هاست که از هیچ کدام از خانه های زرد و آبی عبور نکردیم.

از خانه های زرد نرفتیم چون اولویت سمت چپ بیشتر از سمت راست بود.

از خانه های سفید رد نشدیم چون اولویت چپ بیشتر از راست بود.

از خانه ها بینفش هم رد نشدیم چون اولویت چپ بیشتر از بالا بود.



اردر ما در این تابع به صورت b^m می باشد، و از جایی که می توان **متوسط** تعداد node های اضافه شده در هر حالت را برابر با 2 دز نظر گرفت، پس اردر ما نمایی می شود.

تابع a_star:

در این تابع ساز و کار مشابه ای با تابع bfs را در پیش گرفتیم فقط با توجه به اینکه هر node ممکن است از طریق چند node عه دیگر به لیست بازمان اضافه شده باشد، در لیست باز هنگام اضافه کردن هر node، کنار آن Node عه parent و فاصله ای که تا آن node داشته است را هم قرار می دهیم.
در هر سری اجرای حلقه، node ای را برای بسط دادن انتخاب می کنیم که آن node کمترین مقدار f را داشته باشد (مجموع تابع هیوریستیک آن نقطه بعلاوه فاصله ای که تا آن لحظه طی کرده است.)

در اینجا بر خلاف الگوریتم bfs دیگر node ها را وقتی که در لیست باز اضافه می کنیم در دیکشنری قرار نمی دهیم، بلکه در هنگام بسط دادن آنها را در دیکشنری قرار می دهیم.(مقدار parent را هم با توجه به نحوه ذخیره سازی هر node در لیست باز داریم)

همچنین در این تابع، آزمون هدف، در لحظه ایجاد است. پس در همان بدنه لحقه در اولین لحظه چک میکنیم که آیا این node ای که بهینه ترین بوده، node عه هدف است یا خیر.

(در این الگوریتم هم با توجه به اینکه از اول لیست شروع به پیدا کردن برای بهترین node می کنیم، پس اولویت ها به ترتیب اضافه شده به لیست می باشد، یعنی دقیقا مشابه الگوریتم bfs)

تابع هیوریستیک:

در این سوال، تابع هیوریستیک را چیزی شبیه به فاصله منهتن فرض کرده ایم. یعنی هیوریستیک هر node برابر است با فاصله طولی و عرضی آن node از node عه goal است. یعنی اختلاف x های دو node بعلاوه اختلاف y های دو node.

بررسی درستی این تابع هیوریستیک در نظر گرفته شده:

۱- قابل قبول بودن:
این شرط برقرار است زیرا عددی که این تابع به ما می دهد، همواره (در بهترین حالت، اگر هیچ block ای جلویمان نباشد) بهینه ترین و کمترین مقدار ممکن است. پس فاصله واقعی همواره بزرگتر مساوی این مقدار می باشد.

۲- سازگاری:
این شرط نیز در این تابع برقرار می باشد. چود در هر صورت هزینه صورت گرفته برای انتقال از یک node به node عه دیگر یک است. حال ما یا در هر مرحله به Nodeعه هدف نزدیک تر می شیم یا دورتر (از لحاظ مختصات)

- اگر نزدیک تر شده باشیم، پس هیوریستیک node عه جدید کمتر است (به اندازه یک واحد) ولی از طرفی هم یک واحد هزینه کرده ایم تا به آن node برسیم. پس این یک واحد نزدیک شدن، با آن یک واحد هزینه خنثی می شود. پس این مقدار نامساوی برقرار می ماند (حالت مساوی رخ می دهد.)
- اگر دور تر شده باشیم، مقدار هیوریستیک مان بیشتر شده(به اندازه یک واحد)، و همچنین از طرفی یک واحد هم هزینه کرده ایم. پس باز هم این نامساوی برقرار می ماند.(در حالت کوچک تر بودن بر قرار می شود.)

پس در هر دو حالت، این نامساوی برقرار می ماند.
پس این شرط برقرار است

پس با توجه به اینکه این دو شرط برقرار هستند، پس این تابع هیوریستیک مورد پذیرش است.

اردر ما باز هم (در بدترین حالت) به صورت نمایی می باشد.

شکل حاصل از اجرای این الگوریتم: