

به نام خدا

گزارش پروژه میانترم درس AP

نام و نام خانوادگی : علی شاهی

شماره دانشجویی : 9623063

استاد درس : دکتر جهانشاهی

ابتدا تابع make\_maze و سپس سه الگوریتم BFS ، DFS ، BIDIRECTIONAL توضیح

داده میشود . سپس توابع گرافیکی گفته میشود .

## تابع make\_maze

این تابع ابتدا برای تعریف maze است .

عملکرد این تابع به این شکل است که ابتدا وکتوری با تعداد سطر و ستون هایی که یوزر داده و خالی است تولید میکنیم . سپس با استفاده از `std::rand()` برای هر درایه ماتریس دو بعدی عددی تولید کرده و باقی مانده آن بر 100 را میگیریم و بعد چک میکنیم که اگر عدد رندوم تولید شده از یک عدد آستانه (اینجا 68 را قرار دادم ) بیشتر شد آن را دیوار (#) قرار میدهیم وگرنه آن را یک قرار میدهیم. عبارت `std::rand(time(0))` برای این است که اعداد رندوم پس از هر بار کامپایل یکسان نباشد.

## الگوریتم DFS

کانستراکتور کلاس DFS به این صورت است که MAZE رندوم تولید شده در تابع MAKE\_MAZE و همچنین آرایه ای شامل 4 ورودی که اطلاعات مختصات ورودی و خروجی در آن قرار دارد را میگیرد. این کلاس دو ورودی دیگر هم دارد . TEMP\_ANS برای ذخیره خانه های اصلی مسیر و COM\_ANS برای ذخیره خانه هایی که چک شده است .

سپس تابع DFS\_MAZE صدا زده شده است در این تابع بعد از PUSH\_BACK کردن ورودی درون دو متغیر گفته شده چک میکنیم که اگر اطلاعات خواسته شده از یوزر نا معتبر است ، پیغامی متناسب با آن چاپ شود. مثلا اگر ورودی داده شده یا خروجی داده شده توسط یوزر دیوار باشد ، پیغام خطایی متناسب با آن چاپ شده است.

سپس تابع CHECK\_MAZE صدا زده میشود و درون این تابع ابتدا آرایه ای برای حرکت به چهار جهت اصلی تعریف میشود . سپس چک میکنیم که اولا اگر مختصات جدید ، دیوار نباشد دوما اگر قبلا چک

نشده باشد و سوما اگر خارج مختصات MAZE (ماتریس) نباشد ، این مختصات جدید داخل دو متغیر TEMP و COMP ذخیره میشود و دوباره تابع CHECK\_MAZE ران میشود .(به عبارتی ما از الگوریتم BACK\_TRACKING استفاده میکنیم .) همچنین اگر هیچ کدام از 4 باری که درون این حلقه میچرخد جزو مختصات متناسب چک شدن نباشد ، باید مختصات داده شده به تابع CHECK\_MAZE از درون متغیر TEMP پاک شود (این یعنی اینکه مثلا مختصات جدید به بن بست خورده و مسیری برای جلو رفتن ندارد پس باید از آن برگشت.)

همچنین شرطی در اول FOR چک میکنیم که اگر متغیر TEMP شامل مختصات خروجی بود یعنی MAZE ما تمام شده و RETURN میکنیم . همچنین متغیری را داخل این IF یک میکنیم که اگر از FOR بیرون آمد نیز با چک کردن این متغیر RETURN کند .

سپس بعد تابع CHECK\_MAZE تابع SHOW را ران میکنیم(میتوانستیم تابع SHOW را در APPEARANCE ران کنیم).

درون تابع SHOW چک میکنیم که اگر اندازه متغیر TEMP صفر بود، باید پیغام خطایی متناسب با آن چاپ شود(این حالت زمانی اتفاق می افتد که مسیری از مختصات شروع تا پایان نداشته باشیم). سپس با حلقه زدن روی متغیر TEMP تمام مسیر را از ابتدا تا انتها چاپ میکنیم همچنین برای نمایش روی ماتریس باید اینگونه عمل کنیم که اگر درایه ماتریس جزو مسیر بود ابتدا پیش زمینه پیکسل آن صورتی شود و سپس چاپ شود در غیر این صورت به صورت عادی چاپ شود . اینگونه میتوانیم مسیر را روی ماتریس (MAZE به ماتریس تشبیه شده) ببینیم .

## الگوریتم BFS

کانستراکتور این الگوریتم شبیه الگوریتم قبلی است . درون این کلاس ابتدا کلاس Node را تعریف میکنیم . کلاس Node شامل دو متغیر که مختصات آن درون maze ، یک متغیر \*parent Node و 1 متغیر وکتور 4 تایی از نوع \*Node برای بچه های آن میباشد همچنین این کلاس دارای کانستراکتوری برای مقدار دهی به Node میباشد.

درون تابع `bfs_maze` ، وکتور `root` را تعریف میکنیم و متغیر `proot` را درون آن میریزیم . متغیر `proot` از نوع `Node*` بوده که در کانستراکتور کلاس مقدار دهی میشود و برابر `Node` شروع میشود سپس بعد از چک کردن صحیح بودن مختصات ورودی و خروجی ، تابع `make_bfs_tree` را میخواند . الگوریتم تابع `make_bfs_tree` به این صورت است که در ابتدا متغیر وکتوری به نام `level` تعریف میکنیم این متغیر در هر بار تکرار این تابع باید یک `level` درخت را بگیرد . مثلاً ابتدا `level` اول که ریشه است را میگیرد و بعد `level` دو که بچه های ریشه است را میگیرد و سپس `level` بعدی بچه های هر کدام از بچه های ریشه را میگیرد الی آخر

حال روی هر کدام از `Node` های هر `level` الگوریتم قبلی را تکرار میکنیم

اگر شرایط صحیح مثل دیوار نبودن یک نود که در الگوریتم قبلی بیان شد ، اینجا نیز برقرار باشد ، یک `Node` به نام `check_Node` ساخته میشود و مقدار `x` و `y` جدید درون آن ریخته شده و پدر آن مقدار دهی شده همچنین این نود `level_node` مورد نظر قرار میگیرد .

درون کلاس متغیری تعریف میکنیم به نام `end_node` که درون این نود مختصات پایانی `maze` ریخته میشود . و درون تابع چک میکنیم که اگر مقدار `x` و `y` `level_node` مورد نظر برابر `x_e` و `y_e` شد آن گاه `break` میکنیم و بیرون `for` هم چک میکنیم که اگر `end_node` مقدار داده شد (یعنی `nullptr`) نبود ، دیگر تابع `make_bfs_tree` تکرار نشود .

بعد از تابع `make_bfs_tree` تابع `path_result` را میخواند .

درون تابع `path_result` چک میشود که اولاً اگر آخرین `Node` متغیر `all_Nodes` مختصات پایانی ما نبود ، پیغام ارور مورد نظر را بدهد سپس تا جایی که `end_node` خالی نشود آن را ابتدا درون متغیر `result_path` (متغیری که مسیر مورد نظر در آن ذخیره خواهد شد) ذخیره میکنیم سپس پدر آن را درون خود `end_node` میریزیم . این کار باعث میشود که الگوریتم `bfs` کوتاه ترین مسیر ممکن را به ما بدهد . سپس درون همین تابع ، تابع `SHOW` را اجرا میکنیم . (می توانستیم درون تابع `APPEARANCE` این تابع `SHOW` را نیز اجرا کنیم.)

تابع `SHOW` این الگوریتم نیز دقیقاً مشابه تابع `SHOW` الگوریتم قبلی است .

## الگوریتم bfs\_bidirectional

طرز کار این الگوریتم به این شکل است که از نود ابتدا و همچنین از نود انتها شروع به پیمایش درخت میکند و زمانی که به نود مشترک رسیدند (اینجا مختصات  $x$  و  $y$  مشابه داشته باشند) یعنی مسیر مورد نظر از نود ورودی به نود خروجی یافت شده است .

فایده این روش این است که تعداد نود کمتری برای level های پسین در نظر گرفته میشود ( در حالت درخت level directional های بعدی تعداد نود زیاد تری نسبت به level های قبلی خواهند داشت . با این الگوریتم از دو درخت همزمان استفاده میکنیم که موجب به کاهش نود ها خواهد شد.)

کلاس bfs\_bidirectional ( bi\_bfs ) از کلاس bfs ارث میبرد . پس ما ابتدا یک متغیر دیگر برای نود ریشه یعنی proot2 (مختصات پایانی در آن قرار خواهد گرفت . ) تعریف میکنیم . همچنین مشابه الگوریتم bfs ولی با این تفاوت که دو level\_node داریم به جای تابع make\_bfs\_tree درون کلاس bfs دو تابع start\_bfs\_tree و end\_bfs\_tree داریم که درون هر کدام از توابع باید تابع دیگری اجرا شود . عملکرد هر کدام از توابع مانند make\_bfs\_tree میباشد فقط در پایان چک میشود که اگر نود مشترک یافت شد ، آن گاه start\_node و end\_node برابر نود مورد نظر قرار گیرد و سپس از دو تابع خارج میشود . تابع path\_result نیز مانند قبل است فقط روی دو نود end\_node و start\_node اجرا میشود (مسیر ها داخل همان وکتور های level\_s\_node و level\_e\_node ریخته میشود . )

در پایان دو مسیر level\_s\_node و level\_e\_node داریم که اولی از نود مشترک تا نود ورودی و دومی از نود مشترک تا نود خروجی میباشد و داخل تابع show با استفاده از دو flag اولین مسیر را با رنگ صورتی و دومین مسیر را با رنگ آبی نشان میدهیم.

## تابع main و تابع APPEARANCE

درون تابع main تابع appearance صدا زده شده است . این تابع برای دادن شکل ظاهری (gui) کد میباشد به این صورت که برای چاپ هر قسمتی از یوزر خواسته میشود از کد های ANSI استفاده شده است . مثلاً برای چاپ عبارت #x\_end اعداد رنگ ها به این صورت قرار داده شده که این عبارت به

رنگ صورتی چاپ شود . سپس این ورودی های انتخاب شده بر اساس اینکه یوزر میخواهد با bfs یا maze ، dfs را حل کند یا اینکه آیا bidirectional اجرا شود یا خیر ورودی ها را به کانستراکتور های هر کدام از الگوریتم ها داده میشود .

در پایان برای نوشتن فایل های makefile و همچنین Dockerfile دقیقا مشابه همان فایل هایی که درون تمرین ها قرار داده میشد استفاده کردم.