به نام خدا

گزارش کار پروژه میانی برنامه نویسی پیشرفته

یاسین دهفولی 9623048

ایده اصلی بنده برای ساخت قالب maze استفاده از یک std::vector دو بعدی می باشد که بصورت یک Grid Maze به نمایش در می آید.نقطه شروع حرکت نقطه 0,0 در وکتور 2 بعدی ما می باشد و مقصد نیز طبعا در سطر آخر ستون آخر قرار خواهد گرفت.

حال باید تعداد معقولی از خانه های Maze را بعنوان مانع در نظر گرفت تا یک maze داشته باشیم ولی نمیتوان بصورت رندوم تعدادی از خانه ها را مانع کرد چراکه ممکن است با این کار مسیری از ورودی به خروجی نداشته باشیم. فلذا باید ابتدا از وجود یک مسیر از ورودی به خروجی مطمئن شویم و سپس تعدادی از خانه هایی که در مسیر وجود ندارند را بصورت رندوم بعنوان مانع در نظر بگیریم. پس وارد اولین و ساده ترین الگوریتم مسیر یابی در گراف ها میشویم.

تمام الگوریتم ها برای graph search استفاده دارند اما maze ما از جنس وکتور دوبعدی می باشد پس باید به منظور تعریف یال ها و راس ها قانونی برای مسیر حرکت در maze داشته باشیم که این قانون ساده به صورت زیر می باشد

هر خانه از این آرایه دو بعدی را راس های گراف در نظر میگیریم که در صورت وجود به 4 جهت اصلی اطرافشان یال دارند.

بعبارتی در maze ما از هر نقطه تنها می توانیم به 4 جهت اصلی بالا پایین چپ راست حرکت کنیم آن هم در صورتی که آن خانه را قبلا طی نکرده باشیم و همچنین از maze خارج نشده باشیم حال میتوانیم الگوریتم ها را روی قالب خودمان پیاده سازی کنیم

DFS

توضیحات راجع به خود الگوریتم در اینجا داده نمی شود و فقط نحوه پیاده سازی مختصر توضیح داده شده است . برای پیاده سازی این الگوریتم از stack استفاده شده است .در

ابتدا خانه 0,0 به صورت std::pair به stack ما push میشود و روی stack قرار میگیرد. سپس در تا زمانی که stack خالی نشده باشد خانه بالایی stack را pop میکنیم .سپس مختصات همسایه های خانه ی pop شده را به stack تعریف شده در الگوریتم push میکنیم تا دوباره pop شوند

* نکته اول این است که این اتفاق در صورت valid بودن (قبلا visit نشده باشد و از maze بیرون نزده باشند) خانه های همسایه رخ می دهد که توسط تابع is_valid چک میشود.

* نکته مهم در push کردن خانه های همسایه این است که باید **بصورت رندوم** در stack پوش شوند تا جهت گیری در هر مرحله متفاوت باشد (با استفاده از یک تابع built-in به نام shuffle در direction array در کد ضمیمه شده است) .

در نهایت در هر مرحله چک میکنیم که آیا خانه فعلی که از stack ما pop شده است خانه مقصد هست یا خیر که در صورت رسیدن به مقصد از الگوریتم خارج شویم و جواب را ذخیره کنیم.

ایده بنده برای ذخیره سازی مسیر یافته شده توسط DFS شماره گذاری خانه های وکتور 2 بعدی می باشد. بدین صورت که اولین خانه ای که از stack ما pop میشود و valid میباشد رادر DFS_maze تعریف شده در کد ، 1 میکنم و خانه ی بعدی را 2 و 3 و 4 و ... شماره گذاری میکنم تا به مقصد برسم. با اینکار تک تک step های برنامه به ترتیب تا رسیدن به مقصد مشخص خواهد شد. نکته ای که باعث مشکل در ساختن maze میشد نداشتن مسیر بود که در اینجا این مشکل حل شدو یک مسیر از نقطه آغاز تا مقصد یافت شد و میتوان سایر خانه ها را بصورت رندوم بعنوان مانع در نظر گرفت.

* طبیعت الگوریتم DFS این است که مسیری را به ما میدهد که تعداد خانه های زیادی از maze را برای رسیدن به مقصد طی میکند (تعداد زیادی از خانه های maze که میتوانستند مانع باشند در مسیر یافته شده هستند و باعث محدودیت زیادی در ساختن maze می شوند). برای رفع این مشکل یک شرط ساده در ساخت maze در نظرگرفته شده است. میدانیم که در یک maze بدون مانع کوتاه ترین مسیر از ورودی به خروجی به

اندازه جمع تعداد سطر ها و ستون های آن میباشد . هدف ما این است که تعداد خانه های طی شده در مسیر به اندازه ی معقولی باشد که برای ساخت maze ما محدودیت زیادی ایجاد نگردد فلذا یک متغیر به نام coef بعنوان ورودی تابع dfs() در کد قرار داده شده است تا این مقدار معقول را فراهم سازد. بعبارتی اگر تعداد خانه های مسیر یافته شده از coef*(maze_rows+maze_columns) بیشتر شد الگوریتم DFS دوباره اجرا شود تا بالاخره مسیری پیدا شود که طول آن از عبارت فوق کمتر باشد و بتوانیم maze را بسازیم.

برای اینکه کاربر آزادی عمل زیادی برای ساختن maze مورد نظرش داشته باشد از یک پارامتردیگر استفاده شده است این پارامتر درصدی است که معین میکند چند درصد از خانه های غیر مسیر maze بعنوان مانع مشخص شوند.کد این بخش در تابع random_choose ضمیمه شده است.

پس اکنون هم maze ساخته شد و هم یک مسیر از طریق الگوریتم DFS از ورودی به خروجی به دست آمد.

BFS

این الگوریتم کوتاه ترین مسیر از ورودی به خروجی را به ما می دهد.شکل و مسیر کلی پیاده سازی الگوریتم BFS پیاده سازی شده مانند DFS است و فقط به جای stack از queue استفاده میشود که ذکر تفاوت آنها خارج از بحث گزارش کارمیباشد .

تفاوت دیگر این الگوریتم با DFS این است که این الگوریتم به جای اینکه خانه به خانه جلو برود لایه لایه پیشروی میکند . همین نکته باعث میشود که ما حتی با شماره گذاری خانه ها در هر مرحله بازهم نتوانیم یک مسیر معین و واضح از ورودی به خروجی داشته باشیم بلکه فقط لایه های maze به ترتیب حرکت به سمت مقصد شماره گذاری شده اند و ما چندین 1 و 2 و ...خواهیم داشت که یافتن مسیر را مشکل مینماید . پس باید راهی برای یافتن مسیر از ورودی به خروجی پیدا کنیم.

کافیست که برای هر کدام از خانه های شماره گذاری شده یک parent تعیین کنیم که بتوانیم بگوییم خانه ی قبلی هر خانه طی شده ازمسیر کدام است. اگر parent هر خانه ازمسیر معین باشد میتوانیم از خانه مقصد شروع کنیم و parent آن را بعنوان خانه قبل از

مقصد در مسیر بشناسیم و همینطور تا رسیدن به نقطه شروع ادامه دهیم تا کل مسیر بدست بیاید.

بدلیل اینکه پیاده سازی بنده ، قرار دادن اعداد بعنوان مسیر میباشد نحوه تعیین **parent** برای هر نقطه که یک عدد است کار دشواری می باشد که در مراحل زیر کاملا توضیح داده شده است.

1- یک **Parent_maze** که وکتور **2**بعدی به ابعاد **maze** میباشد میسازیم.

2- تمام خانه های آن را بصورت سطری با اعداد **0** تا **1-(rows*columns)** شماره گذاری میکنیم(**point**).

3-توابع **point_to_cordinates** و **coordinates_to_point** را برای تبدیل هرشماره به مختصات و بالعکس می نویسیم (در کد ضمیمه شده است)

4-یک وکتور تک بعدی به نام **parent_vector** و با ابعاد **rows*columns** میسازیم.

5- حالا درهر خانه ای که بودیم ، آن را بعنوان **parent** برای تمام **4** جهت اصلی اش مشخص میکنیم چرا که از این خانه فعلی به آن جهت ها رفته اند.

6-برای مشخص کردن اینکه کدام خانه از **parent_maze** و **parent_maze** استفاده شده است فرض کنیم یک **maze** با ابعاد **10** در **10** داریم که در خانه **1,1** آن هستیم

مقدار خانه ی **(1,1)** در وکتور **2** بعدی **parent_maze** را بدست می آوریم (مثلا نام آنرا **C** می گذاریم). حال باید **parent** تمام خانه های بالایی ، پایینی ، چپی ، راستی خانه **(1,1)** را ، خود خانه ی **(1,1)** قرار دهیم .

پس ازآن مقدار خانه های **(0,1)** و **(1,0)** و **(2,1)** و**(1,2)** که جهت های اطراف **(1,1)** هستند را در وکتور **2** بعدی **parent_maze** بدست می آوریم (مثلا نام آنها را **point** می گذاریم)

<u>حال مقدار خانه ی point اُم parent_vector را C میگذاریم</u>

Parent_vector[(Parent_maze[0][1])] =Parent_maze[1][1]

Parent_vector[(Parent_maze[1][0])] =Parent_maze[1][1]

Parent_vector[(Parent_maze[2][1])] =Parent_maze[1][1]

Parent_vector[(Parent_maze[1][2])] =Parent_maze[1][1]

کد بالا مقداری پیچیده است و ایده شخصی بنده برای تعیین parent ها در مسیربوده است ☺ برای درک بهتر و تصویری موارد فوق به پاورپوینت مراجعه فرمایید .

Bidirectional search

این الگوریتم که در بخش امتیازی صورت سوال بود برای کاهش زمان و محاسبات درمسیریابی گراف ها استفاده میگردد.در این الگوریتم کافیست با یکی از الگوریتم های BFS,DFS از نقطه شروع به سمت نقطه پایان حرکت کنیم (مسیر رفت) و همزمان از نقطه مقصد به سمت نقطه آغاز حرکت کنیم(مسیر برگشت) و هر جا که مسیر رفت و مسیر برگشت باهم برخورد کردند الگوریتم به پایان میرسد و همان مسیر BFS یا DFS (بسته به اینکه از کدام روش استفاده شده است) بدست خواهد آمد.

برای پیاده سازی این الگوریتم بنده از BFS استفاده کرده ام ولی متاسفانه پیاده سازی الگوریتم های BFS,DFS بنده به گونه ای بوده است که امکان اجرای همزمان آنها روی یک maze وجود نداشت و باید تمام کدهای قبلی الگوریتم ها را کاملا تغییر میدادم که فرصت نکردم . فلذا مجبور شدم که روی maze یکبار ()bfs بزنم و یکبار ()Backward_bfs بزنم

مسیر بدست آمده از ()bfs در یک وکتور به نام bfs_route ذخیره شده است و مسیر بدست آمده از()Backward_bfs در یک وکتور به نام Backward_bfs_route ذخیره شده است . روی bfs_route به سمت انتها حرکت می کنیم و همزمان روی Backward_bfs_route به سمت ابتدا حرکت میکنیم و به محض اینکه نقطه تلاقی آنها را پیدا کردیم آن را ذخیره میکنیم .نیمه اول مسیر را bfs_route معین میکند (ازابتدا تا نقطه تلاقی) و نیمه دوم مسیر را Backward_bfs_route معین میکند (ازنقطه تلاقی تا انتها).