به نام خدا

پروژه میانی درس برنامه نویسی پیشرفته

عنوان پروژه : تولید یک ماز تصادفی و حل آن با الگوریتم های BFS و DFS

نام استاد : جناب آقای دکتر جهانشاهی

نام گرد آورنده : مارال مرداد

شماره دانشجویی : ۹۷۲۳۱۴۸

بهار ۱۴۰۰

مقدمه

در این پروژه هدف ساخت یک ماز تصادفی با ابعادی دلخواه و حل آن با استفاده از دو الگوریتم معروف BFS و DFS می باشد و همچنین می بایست مراحل حل ماز نیز به صورت گرافیکی به کاربر نمایش داده شود.

در بخش اول این گزارش به چگونگی ساخت ماز تصادفی پرداخته شده است، در بخش دوم حل این ماز به وسیله الگوریتم BFS صورت گرفته است و همچنین در بخش سوم حل ماز به روش DFS بیان شده است.

به منظور نمایش مراحل حل ماز از دو روش مجزا استفاده شده است در روش اول که در بخش چهارم توضیح داده می شود، از محیط کنسول و کتابخانه ی ncurses استفاده شده است که محیط گرافیکی بسیار ساده ای را فراهم می کند.

در بخش نهایی تمامی الگوریتم ها به Qt framework منتقل شده و از Qt برای ساخت یک محیط گرافیکی حرفه ای و شکیل تر استفاده شده است.

بخش اول – ساخت ماز تصادفی

برای ساخت یک ماز تصادفی ابتدا ابعاد ماز را مشخص کرده و سپس با در نظر گرفتن این ابعاد یک ماز با تمامی دیوار ها ساخته می شود. یعنی هر خانه (cell) در هر طرف دارای دیوار می باشد. یعنی هر خانه دارای ۴ دیوار بالا، پایین، چپ و راست می باشد.

حال به وسیله ی الگوریتم زیر دیوار های هر cell را به گونه ای حذف می کنیم تا یک ماز معتبر(مازی که از ورودی به خروجی حتما مسیر داشته باشد) ساخته شود.

توضیح الگوریتم:

۱) تمامی cell ها در ابتدا به صورت not visited تعریف می شوند.
۲) تمامی cell ها در ابتدا هر ۴ دیوار را دارند.
۳) از خانه start شروع می شود.
۴) یکی از همسایگی هایی که قبلا visit نشده به صورت تصادفی انتخاب می شود.
۵) به آن خانه رفته و دیوار بین cell قبلی و cell جدید انتخاب شده برداشته می شود.
۶) اگر تمامی همسایگی ها در مرحله ۴، visit شده بودند، یک خانه به عقب می رویم.
۷) مراحل ۴، ۵ و ۶ را تا زمان visit شدن تمامی cell ها ادامه می دهیم.

در این روش ساخت ماز اثبات می شود که بین هر دو خانه ی منتخب در ماز حتما یک مسیر وجود دارد.

برای پیاده سازی این الگوریتم از دو کلاس cell و maze استفاده شده است. cell هر یک از خانه های ماز را می سازد و maze الگوریتم بالا را برای تولید ماز پیاده سازی می کند.

توضیح کلاس Cell:

این کلاس دارای member variable های زیر می باشد:

- is_visited مشخص می کند خانه قبلا visit شده یا خیر.
- row و column شماره ردیف و ستون cell در ماز را مشخص می کند.
- wall_up/down/right/left وجود یا عدم وجود دیوار های دور cell را مشخص می کنند.

توضیح کلاس Maze:

این کلاس دارای member variable و method های زیر می باشد:

- path ، یک stack از نوع *Cell می باشد. با حرکت در ماز به وسیله ی الگوریتم توضیح داده شده با رسیدن به هر خانه، آن خانه را در stack نگه می دارد. و در صورت نیاز برای برگشت به خانه های قبلی با توجه به الگوریتم توضیح داده شده از آن استفاده می شود.
- Maze یک vector دو بعدی از *Cell ها است که در واقع خانه های ماز را مشخص می کند.
- تابع check_cell_neighbors، این تابع همسایگی های یک Cell را پیدا کرده و آن هایی که visit نشده اند را ذخیره می کند.
- تابع find_nextcell، این تابع از میان همسایگی های تعیین شده توسط تابع بالا، یکی را به صورت تصادفی انتخاب می کند.
- تابع remove_wall، این تابع با گرفتن دو cell دیوار بین آن ها را تشخیص داده و دیوار بین آن ها را حذف می کند.
- تابع generate_maze، به وسیله ی method های بالا الگوریتم توضیح داده شده را پیاده سازی میکند.

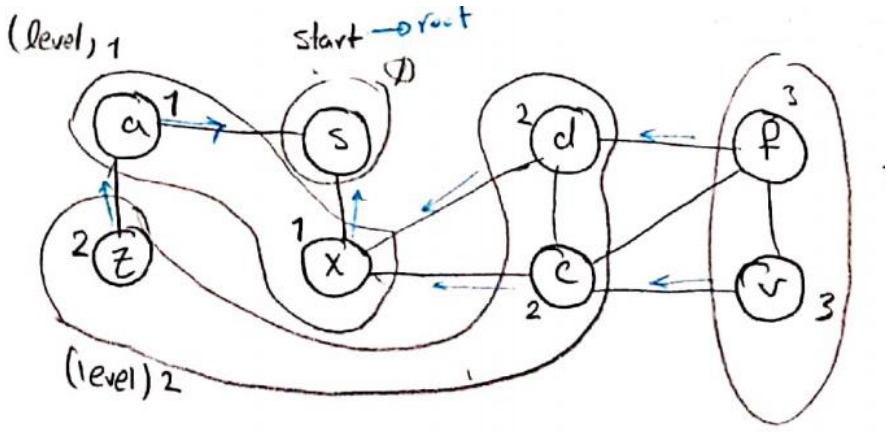
```
void Maze::generate_maze()
{
    while (true) {
        current->set_visited(true);
        check_cell_neighbors();
        Cell* next = find_nextCell();

        if (next != nullptr) {
            next->set_visited(true);
            path.push(current);
            remove_wall(current, next);
            current = next;
        }
        else if (path.size() > 0) {
            current = path.top();
            path.pop();
        }
        else if (path.size() == 0) {
            break;
        }
    }
}
```

شکل ۱- الگوریتم تولید ماز تصادفی

بخش دوم – توضیح الگوریتم BFS

در این بخش با توجه به مراحل زیر می خواهیم ماز را به وسیله ی الگوریتم BFS یا Breadth-first-search حل کنیم.

الگوریتم BFS یک الگوریتم پیمایش ماز است که با مثال زیر به شرح آن می پردازیم:



شکل۲ – گرافی که با الگوریتم BFS پیمایش شده است.

در این گراف راس s را به عنوان نقطه ی شروع انتخاب کرده و این node به عنوان level0 در نظر گرفته می شود. در ابتدا node های متصل به این راس را پیدا می کند که در این جا a و x هستند این دو node به عنوان level1 مشخص می شوند و همچنین s به عنوان parent برای a و x در نظر گرفته می شود. سپس این الگوریتم node هایی که به level1 متصل هستند را به عنوان level2 مشخص می کند و node های level1 را، بر اسات داشتن یا عدم داشتن مسیر، به عنوان parent برای node های level2 مشخص میکند. و همین روند را تا وقتی تمام node ها را پیمایش کند ادامه می دهد.

حال با توجه به parent های محاسبه شده می توان از هر نقطه ای مسیر را به نقطه ی شروع پیدا کرد و این مسیر کوتاه ترین مسیر ممکن بین نقطه ی شروع و نقطه ی مورد نظر می باشد.

به منظور حل ماز کلاس MazeSolver تعریف شده که در آن هر دو الگوریتم BFS، که در این بخش به توضیح آن پرداخته میشود، و الگوریتم DFS که در بخش بعدی شرح داده می شود، پیاده سازی شده اند.

این کلاس یک ماز از نوع *Maze دارد و خانه ی ابتدایی آن یعنی maze[0][0] و خانه ی انتهایی آن یعنی maze[end][end] در آن از نوع *Cell تعریف شده است.

Member variable و method های لازم برای برای BFS:

- level، یک map از *Cell به int است که به هر خانه ی ماز level توضیح داده شده در بالا را map میکند.
- parent، یک map از *Cell به *Cell است که طبق توضیحات بالا به هر خانه ی ماز parent آن را map می کند.
- تابع adjacency، این تابع یک *Cell ورودی می گیرد بر اساس وجود یا عدم وجود دیوار همسایه های آن را مشخص می کند.
- تابع BFS_algorithm، در این تابع الگوریتم توضیح داده شده در بالا برای BFS پیاده سازی شده است و به این ترتیب با پیمایش کل ماز متغیر های level و parent مقدار دهی می شوند.
- تابع BFS_path، این تابع با پیمایش بر روی متغیر parent مسیر متصل کننده end به start را پیدا می کند.

```cpp
void MazeSolver::BFS_algorithm()
{
    std::vector<const Cell*> frontier;
    level[start] = 0;
    parent[start] = nullptr;
    int i = 1;
    frontier.push_back(start);
    while(!frontier.empty())
    {
        std::vector<const Cell*> next;
        for(auto u : frontier)
        {
            for(auto v: adjacency(u))
            {
                if(level.find(v) == level.end())
                {
                    level[v] = i;
                    parent[v] = u;
                    next.push_back(v);
                }
            }
        }

        frontier.clear();
        frontier = next;
        i += 1;
    }
}
```

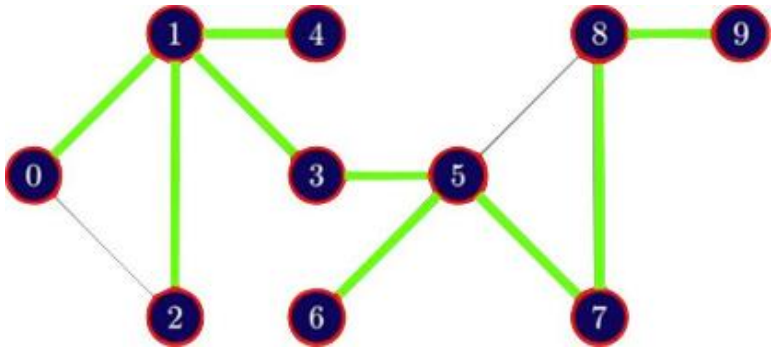
```cpp
std::vector<const Cell*> MazeSolver::BFS_path()
{
    BFS_algorithm();
    std::vector<const Cell*> solve_path;
    solve_path.push_back(end);

    while(parent[end] != nullptr)
    {
        end = parent[end];
        solve_path.push_back(end);
    }

    std::reverse(solve_path.begin(), solve_path.end());
    return solve_path;
}
```

شکل۳ – پیاده سازی الگوریتم BFS و یافتن مسیر

بخش سوم – توضیح الگوریتم DFS

در این بخش با توجه به مراحل زیر می خواهیم ماز را به وسیله ی الگوریتم DFS یا Depth-first-search حل کنیم.

الگوریتم DFS یک الگوریتم پیمایش ماز است که با مثال زیر به شرح آن می پردازیم:



شکل۴ – گرافی که با الگوریتم DFS پیمایش شده است.

در این گراف راس 0 به عنوان شروع در نظر گرفته می شود، برای پیمایش گراف و با توجه به یال های متصل به راس 0 دو انتخاب موجود است، راس 1 و راس 2. در صورت انتخاب راس 1، برای ادامه سه انتخاب وجود خواهد داشت، راس های 2، 3 و 4. اگر از بین این ها 2 انتخاب شود از راس دو برای ادامه مسیری وجود نخواهد داشت لذا یک خانه به عقب برگشته و دوباره از راس یک بین خانه های موجود که قبلا انتخاب نشده اند، انتخاب جدیدی صورت می گیرد. این روند را ادامه می یابد تا تمام گراف پیمایش شود.

Member variable و method های لازم برای برای DFS:

- Marked، این متغیر یک vector دو بعدی به ابعاد ماز می باشد که هر عنصر این ماتریس، نشان دهنده ی آن است که آیا خانه ی متناظر با این عنصر در ماز تا به حال از آن عبور صورت گرفته است یا خیر.
- Visit، یک vector از نوع *Cell می باشد که ترتیب عبور از خانه های ماز را توسط الگوریتم بالا در خود ذخیره می کند.
- تابع adjacency، توضیحات این تابع در قسمت BFS آورده شده است.

- تابع DFS_algorithm، در این تابع الگوریتم توضیح داده شده در بالا به صورت recursive برای DFS پیاده سازی شده است و به این ترتیب با پیمایش ماز تا رسیدن به خانه ی end، متغیر های visit و marked را مقدار دهی می کند.

طبق الگوریتم بالا که الگوریتم پیاده سازی DFS است در صورت رسیدن به نقاط بن بست الگوریتم دارای پرش می شود به عنوان مثال از خانه ی ۹ به ۶ می رود. این اتفاق در پیمایش یک گراف مشکلی به وجود نخواهد آورد اما در مسئله ی ماز امکان پرش در مسیر وجود ندارد. برای حل این مشکل می بایست راه حلی اندیشیده شود تا این پرش در مسیر دیده نشود به این منظور خط هایلایت شده که در شکل ۵ مشاهده می شود به این تابع اضافه شده است که باعث دیده شدن راه های برگشت در متغیر visit می شود.

- تابع DFS_path، با توجه به توضیحات قسمت بالا این الگوریتم علاوه بر مسیر رفت مسیر برگشت از خانه های ماز را نیز در بر میگیرد لذا کافی است با پیمایش بر روی متغیر visit مسیر start به end را بیابیم.

```
void MazeSolver::DFS_algorithm(const Cell* cell)
{
    visit.push_back(cell);
    marked[cell->get_row()][cell->get_column()] = true;
    if(cell == end)
        return;
    std::vector<const Cell*> neighbors = adjacency(cell);
    for(const auto  c : neighbors)
    {

        if( marked[c->get_row()][c->get_column()] == false)
        {
            DFS_algorithm(c);
            visit.push_back(cell);

        }
    }
}

 std::vector<const Cell*> MazeSolver::DFS_path()
 {
     std::vector<const Cell*> solve_path;
     DFS_algorithm(start);
     for(auto cell : visit){
         solve_path.push_back(cell);
         if(cell == end)
             break;

     }
```

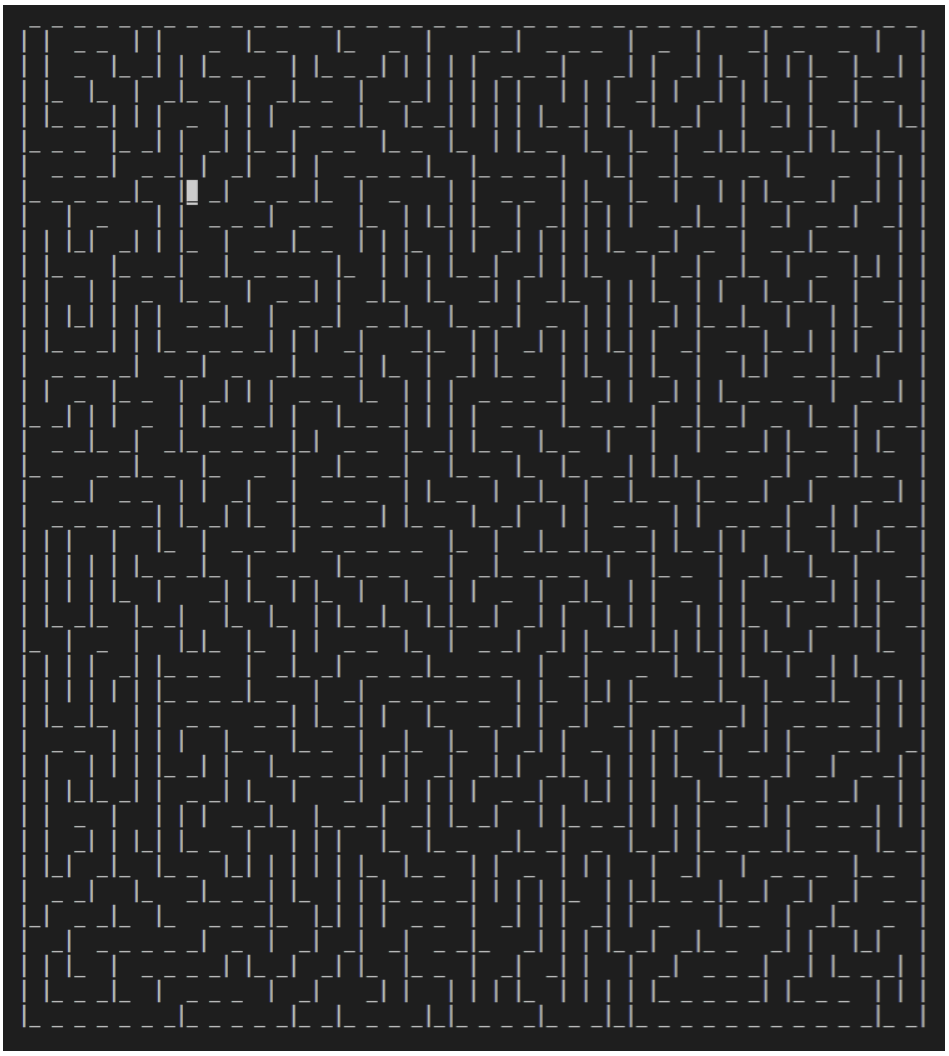
شکل ۵ – پیاده سازی الگوریتم DFS و یافتن مسیر

بخش چهارم – نمایش گرافیکی توسط کتابخانه ncurses

این کتابخانه یک ابزار برای نمایش محیط گرافیکی ساده در بخش کنسول می باشد. ابتدا این کتابخانه را توسط دستور زیر در داکر نصب میکنیم.

```
apt-get install libncurses5-dev libncursesw5-dev
```

حال با هر بار حرکت در ماز صفحه ی کنسول را پاک کرده و ماز را توسط کاراکتر های _ و | رسم می کنیم و curser کنسول را در محلی که در ماز قرار داریم حرکت می دهیم.

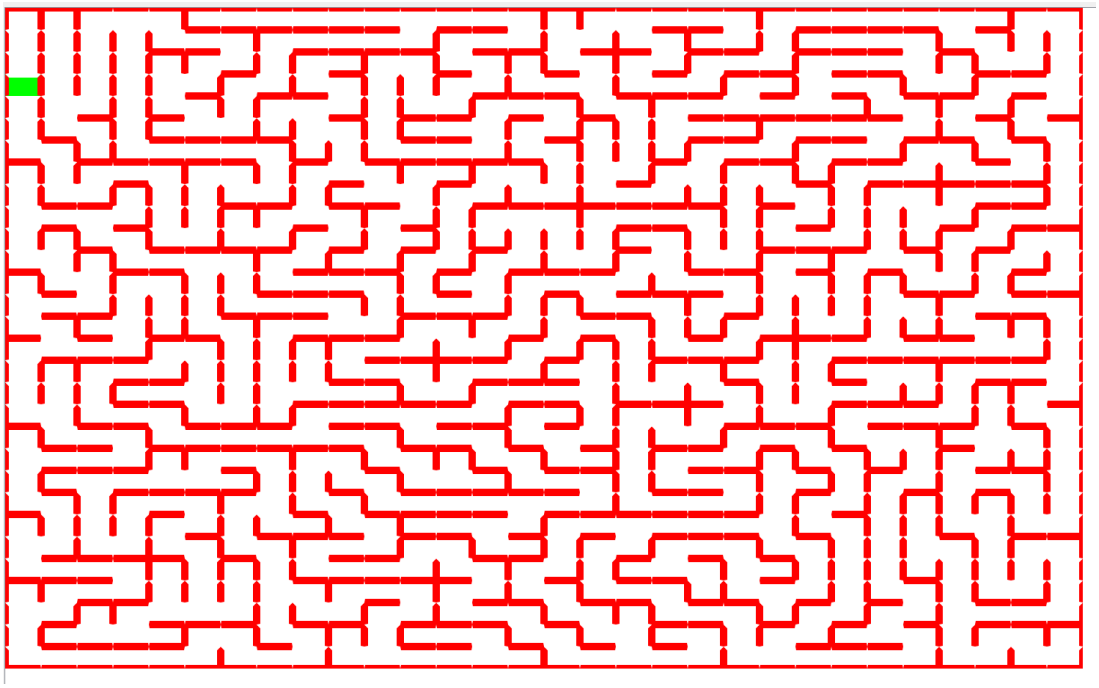


شکل ۶ – گرافیک ماز توسط ncurses

بخش پنجم – نمایش گرافیکی توسط فریم ورک Qt

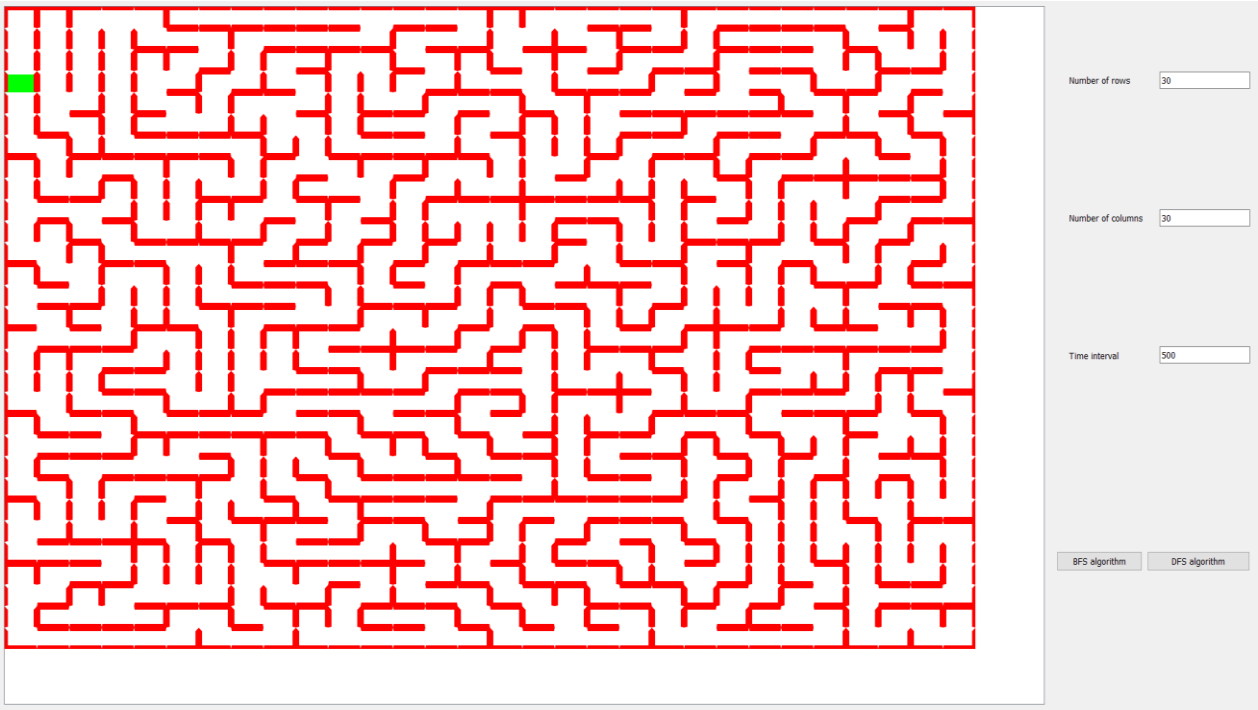
در این قسمت قصد بر آن بود تا یک محیط گرافیکی حرفه ای و شکیل تر برای ماز طراحی شود. بدین منظور از فریم ورک Qt استفاده شده است.

برای نشان دادن هر cell از یک QPushbutton که قابلیت کلیک آن disable شده استفاده شده است و دیوار های این cell توسط border های این button نمایش داده می شود. از کنار هم قرار گرفتن تمامی cell ها در یک QTableWidget ماز نهایی تکمیل می شود. همچنین محل قرارگیری در ماز نیز با تغییر رنگ button آن محل به رنگ سبز مشخص می شود.



شکل ۷ – نمایش ماز در محیط Qt

در کنار نمایش ماز این قابلیت نیز به کاربر داده می شود تا بتواند مشخصات ماز را به صورت real-time تغییر دهد و نتیجه ی آن را مشاهده کند. بدین منظور در کنار ماز محیطی در نظر گرفته شده تا به وسیله ی کلاس های QLineEdit و QPushButton کاربر بتواند مشخصات دلخواه را انتخاب و اعمال کند.



شکل ۸ – نمایش ماز به همراه صفحه ی setting

تمامی کد ها و الگوریتم های پیاده سازی شده در این لینک گیت هاب قابل مشاهده می باشد.