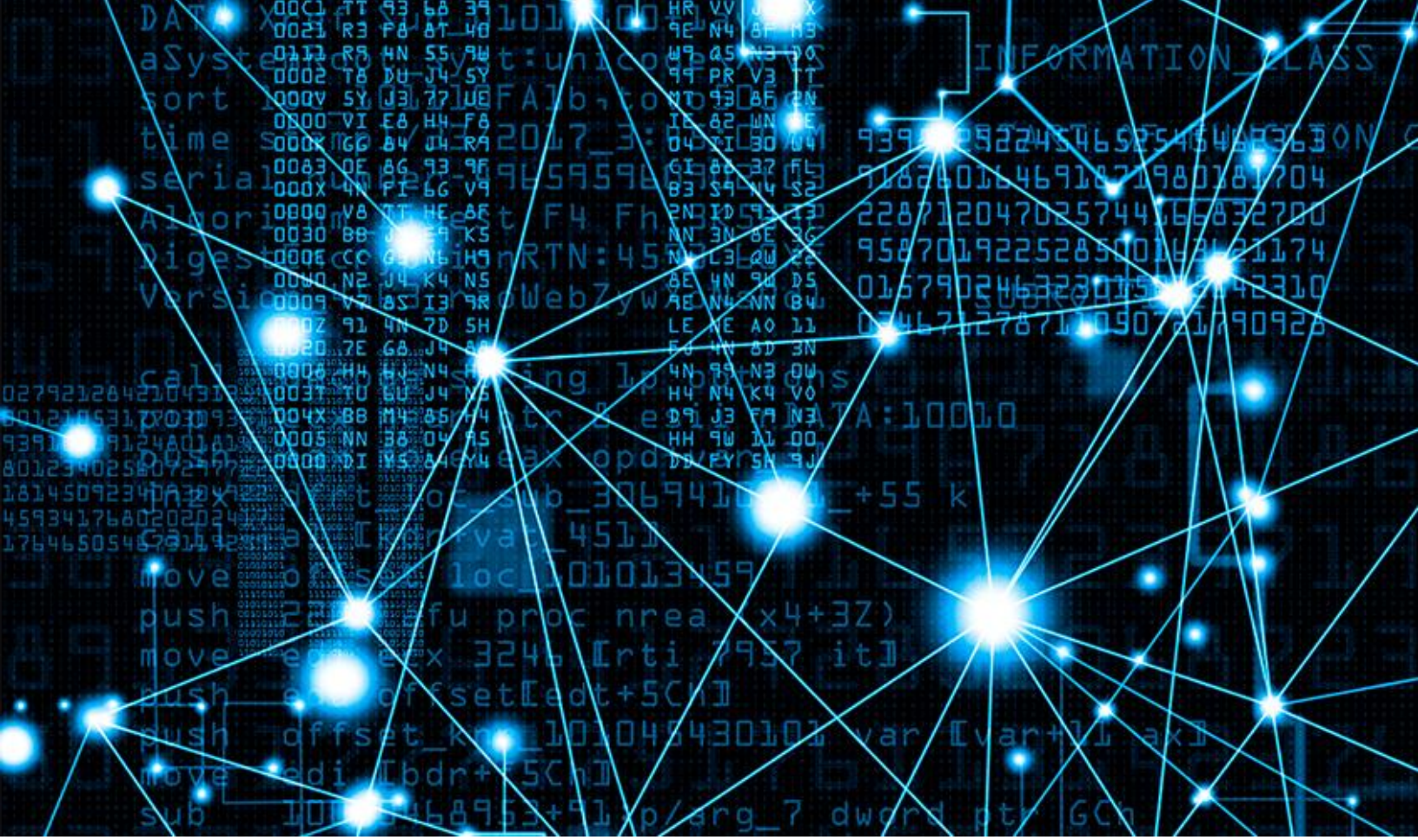


گزارش پروژه الگوریتم

محمدرضا اخگری زیری

۹۶۳۱۰۰۱

زمستان ۹۸

MAD MAZE

توضیح بازی:

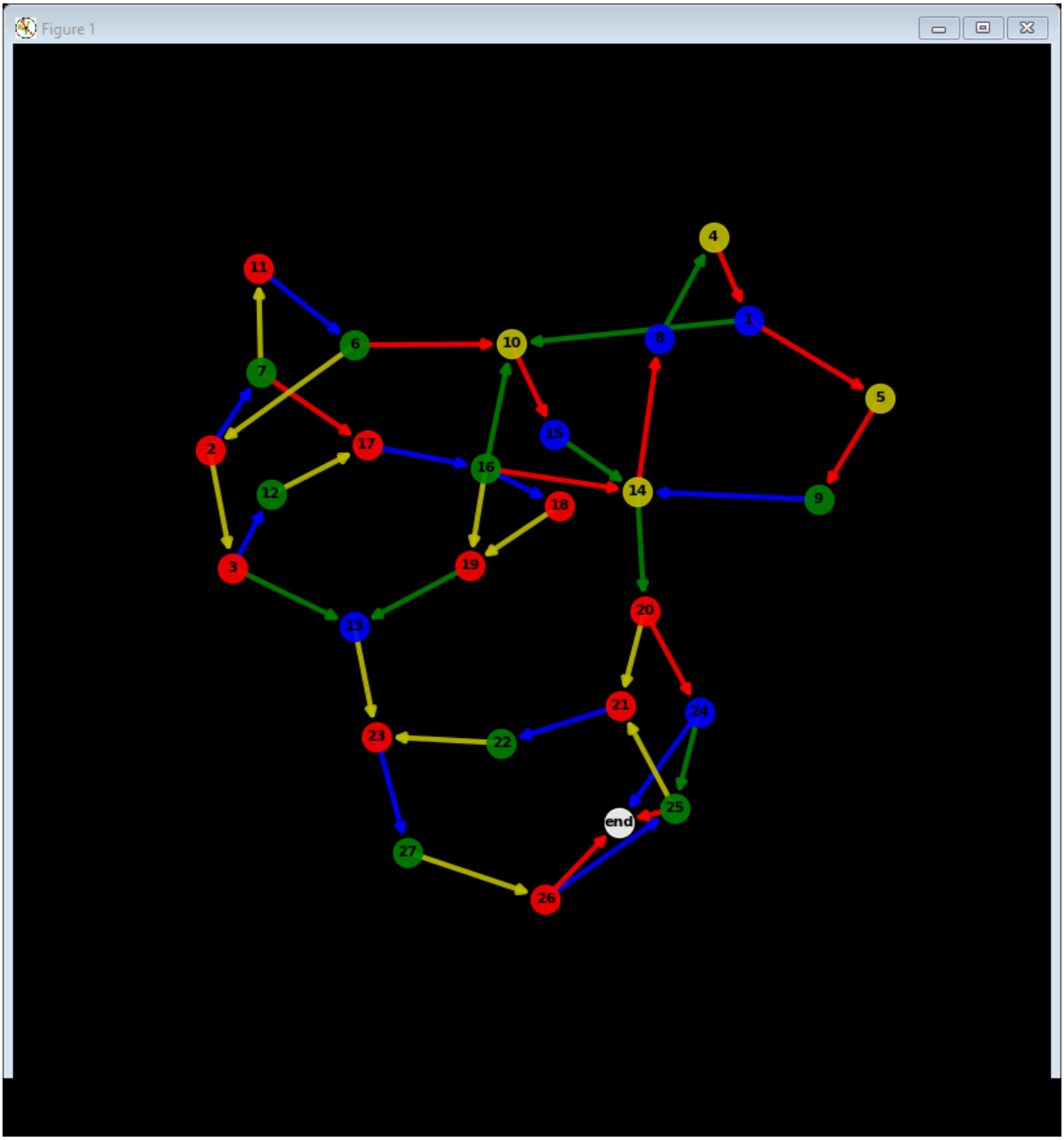
در این پروژه قصد اتمام بازی mad maze را داریم. شرح بازی بدین صورت است که دوشخصیت این بازی (L, R) در دو خانه قرار دارند و هدف رساندن یکی از این افراد به خانه هدف است. در هر مرحله هر بازیکن میتواند به خانه ای برود که دارای شرط زیر باشد:

باید راه رسیدن به ان خانه همرنگ با خانه بازیکن دیگر باشد.

یعنی اگر بازیکن اول در خانه قرمز باشد و بازیکن دوم در خانه آبی باشد، بازیکن قرمز به خانه ای از همسایه ها میتواند برود که دارای یالی به رنگ آبی باشد.

مدل سازی:

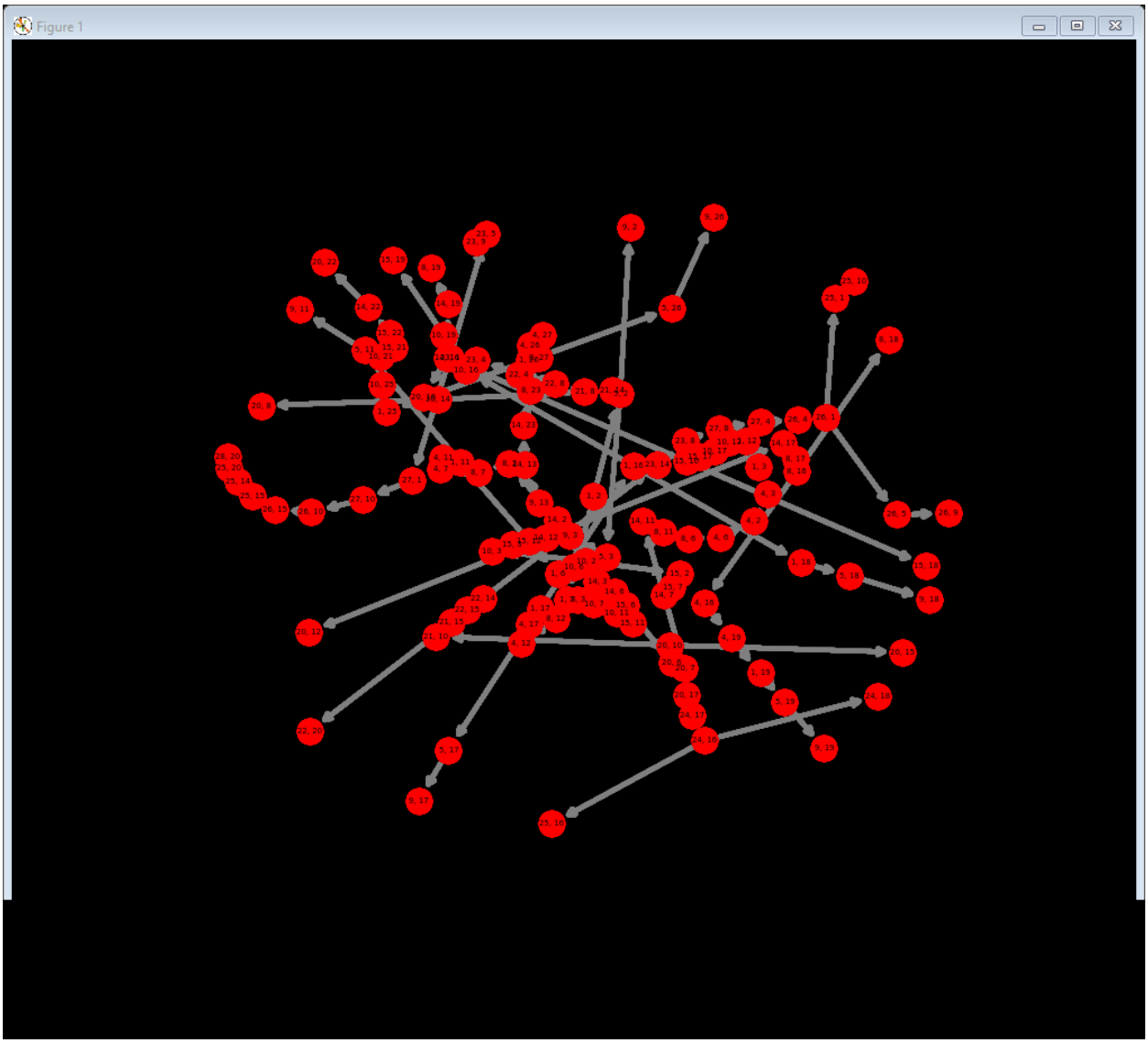
در ابتدا گرافی شامل خانه ها و رنگ آن خانه ها را با کتابخانه networkx در زبان python میسازیم. شکل این گراف در پایین آمده است.



بعد از آن اقدام به ساختن گرافی درختی از گراف اولیه میکنیم (RUG)[1]. بدین شرح که هر نود آن دارای محل دو بازیکن است.

برای ساخت این گراف کافیست تا تمام حالت های قابل ایجاد را به صورت درخت در بیاوریم و یال ها بدین صورت است که از کدام حالت به این حالت آمده است.

نمودار این گراف در پایین آمده است، البته چون تعداد آن زیاد است، تصویر ناواضح است.



[1] Refurbished Ultimate Graph

حال کافیست تا پیمایش bfs را برروی این گراف پیاده کنیم. این پیمایش به ما مسیر تا پایان را میدهد، البته چون پیمایش کرده است نودهای اضافه نیز تولید شده است. پیمایش با استفاده از تایع bfs_predecessors در کتابخانه networkx انجام شده است.

نتیجه پیمایش bfs به شکل زیر میباشد. در این متن (a,b) بدین معناست که از حالت b به حالت a رفته است.

(نتیجه را همراه با پروژه ارسال کردم که قابل خواندن باشد)

حال کافیست تا راه برگشت برای این مسیر را بیابیم.

بدین صورت عمل میکنیم که پیمایش رو به عقب روی نتیجه bfs میزنیم و اگر به خانه مقصد رسیدیم پیدا میکنیم که از کدام خانه به اینجا رسیده‌ایم و این بار شروع به پیدا کردن خانه ای میکنیم که با آن به خانه دیگر رسیده ایم و این کار را تا انتها انجام میدهیم.

این کار به ما نتیجه زیر را میدهد.

L 7 // Lucky moves to G

R 10 // Rocket moves to J

L 11 // Lucky moves to K

R 15 // Rocket moves to O

L 6 // Lucky moves to F

R 14 // Rocket moves to N

L 2 // Lucky moves to B

R 8 // Rocket moves to H

L 7 // Lucky moves to G

R 4 // Rocket moves to D

L 11 // Lucky moves to K

R 1 // Rocket moves to A

L 6 // Lucky moves to F

R 10 // Rocket moves to J

L 2 // Lucky moves to B

R 15 // Rocket moves to O

L 7 // Lucky moves to G

R 14 // Rocket moves to N

L 11 // Lucky moves to K

R 8 // Rocket moves to H

L 6 // Lucky moves to F

R 4 // Rocket moves to D

L 2 // Lucky moves to B

L 3 // Lucky moves to C

R 1 // Rocket moves to A

L 12 // Lucky moves to L

R 10 // Rocket moves to J

L 17 // Lucky moves to Q

R 15 // Rocket moves to O

L 16 // Lucky moves to P

R 14 // Rocket moves to N

R 20 // Rocket moves to T

L 14 // Lucky moves to N

R 21 // Rocket moves to U

L 8 // Lucky moves to H

R 22 // Rocket moves to V

L 4 // Lucky moves to D

R 23 // Rocket moves to W

L 1 // Lucky moves to A

R 27 // Rocket moves to AA

L 10 // Lucky moves to J

R 26 // Rocket moves to Z

L 15 // Lucky moves to O

R 25 // Rocket moves to Y

L 14 // Lucky moves to N

L 20 // Lucky moves to T

R 28 // Rocket moves to end

برای تبدیل کردن این اعداد به حروف الفبا از باقیمانده آنها به ۲۶ استفاده کردیم.

پیمایش به صورت bfs در گراف RUG در صورت بودن جواب نتیجه را میدهد، چون گراف RUG تمام حالت ها را برای ما ایجاد کرده است، در پیاده سازی این گراف درصورت رسیدن به حالت تکراری ادامه مسیر را نخواهیم رفت.

در صورت وجود نداشتن جواب هم در مرحله آخر که روی پیمایش bfs بازگشت میزنیم به حالت نهایی نمیرسد و به ما اعلام میکند مسیری وجود ندارد.

کد:

برای اجرای کد نیاز به کتابخانه های networkx و mathplotlib دارید.

در صورت داشتن فایل ورودی کافیست تا کد را به صورت زیر اجرا کنید.

python main.py <file_path>

که file_path آدرس نسبی فایل ورودی نسبت به محل قرارگیری main.py است. در صورت وارد نکردن این آدرس فایل input.txt را به صورت ورودی فرض میکند/

(فایل input.txt همان ورودی سوال است که همراه پروژه قرار دارد)

برای نگهداری حالت R و S نیز از ساختمان داده ی namedtuple در پایتون استفاده شده است.

بخش امتیازی:

در این پروژه نتایج به صورت گرافیکی رسم شده است. ضمنا پیاده سازی توابع با استفاده از کتابخانه های پایتون بوده است که در گزارش ذکر شده است.

روش دیگری نیز برای حل این سوال وجود داشت که به شرح زیر بود:

میتوانستیم از dfs استفاده کنیم که در این حالت پیاده سازی backtrack انجام میشد.

یعنی در هر حالت به یکی از حالت هایی که میتوانستیم برویم، میرفتیم و دوباره الگوریتم را فراخوانی میکردیم. اگر به حالت تکراری یا حالت بن بست میرسیدیم به مرحله قبل بازمیگشتیم و راه دیگری را امتحان میکردیم.