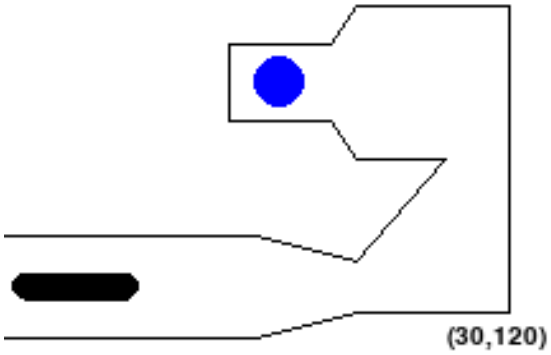


CS440/ECE448 Fall 2021

Assignment 2: Configuration Space Planning

Due date: Wednesday, September 22 11:59pm

In this assignment you will write code that transforms a shapeshifting alien path planning problem into a configuration space, and then search for a path in that space. See the Robotics sections of the lectures for background information.

General guidelines

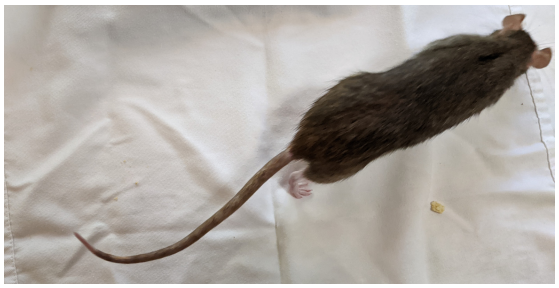
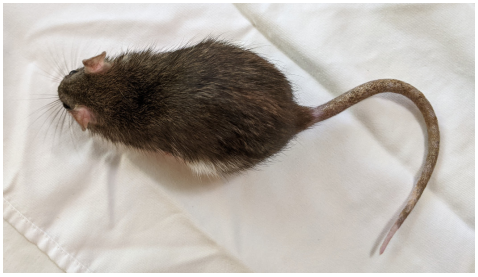
Basic instructions are the same as in MP 1. Specifically, you should be using Python 3.8 or 3.9 with pygame installed, and you will be submitting the code to Gradescope. Your code may import modules that are part of the standard python library, and also numpy and pygame.

For general instructions, see the main MP page and the syllabus.

You will need to adapt your bfs code from MP 1.

Problem Statement

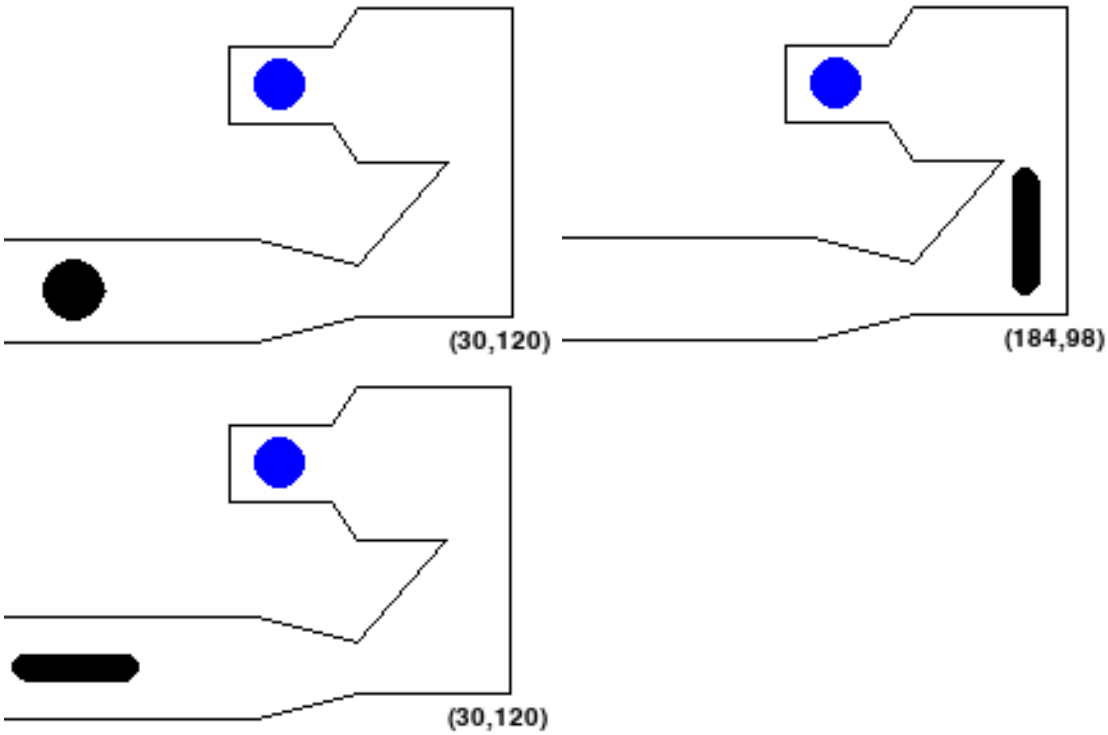
Many animals can change their aspect ratio by extending or bunching up. This allows animals like cats and rats to fit into small places and squeeze through small holes.

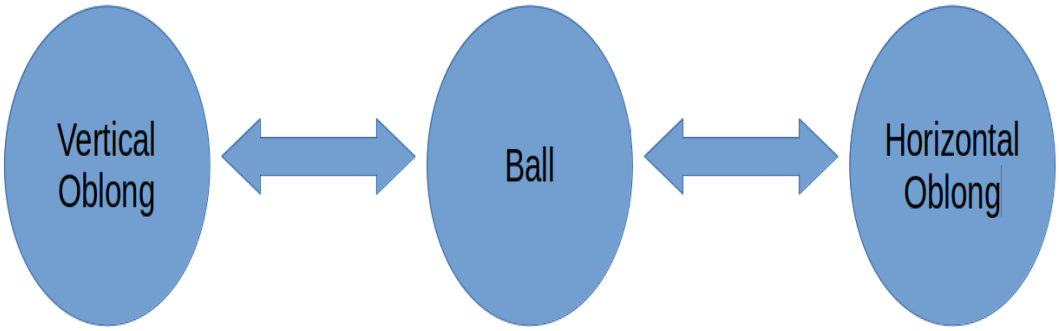



You will be moving an alien robot based on this idea. Specifically, the robot lives in 2D and has three degrees of freedom:

- It can move the (x,y) position of its center of mass.
- It can switch between three forms: a long horizontal form, a round form, and a long vertical form.

Notice that the robot cannot rotate. Also, to change from the long vertical form to/from the long horizontal form, the robot must go through the round form, i.e., it cannot go from its vertical oblong shape to its horizontal oblong shape directly.





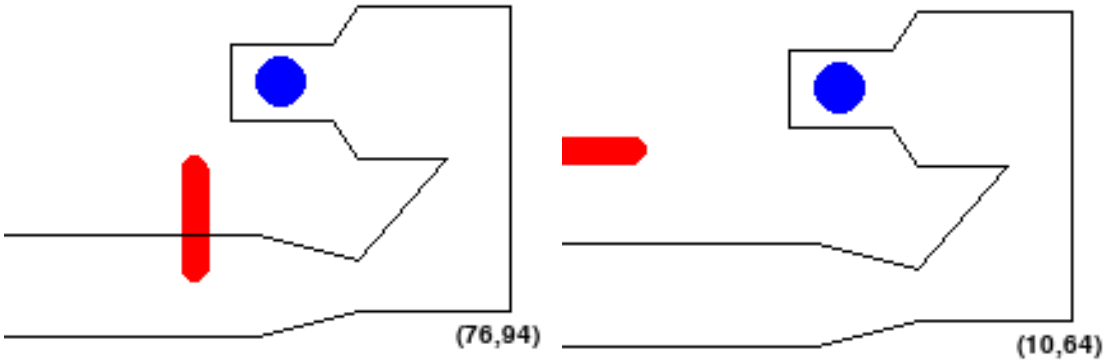
The round form is a disk (or a filled circle). The horizontal or vertical long form is a "sausage" shape, defined by a line segment and a distance "d", i.e., **any point within a given distance "d" of the LINE SEGMENT between the alien's head and tail is considered to be within the alien** .

For each planning problem you will be given a 2D environment specified by:

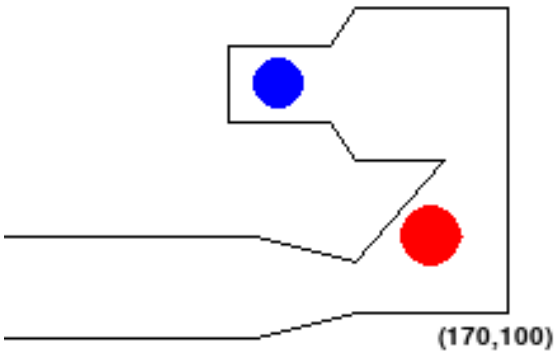
- The size of the workspace.
- The widths of the long and round forms.
- The starting (x,y) center-of-mass position and shape of the alien.
- A list of goals, each of which is defined by disk described by an (x,y) position and a radius.
- A set of obstacles, each of which is a line segment.

You need to find a shortest path for the alien robot from its starting position to one of the goal positions.

The tricky bit is that the alien may not pass through any of the obstacles. Also, no part of the robot should go outside the workspace. So configurations like the following are **NOT** allowed:

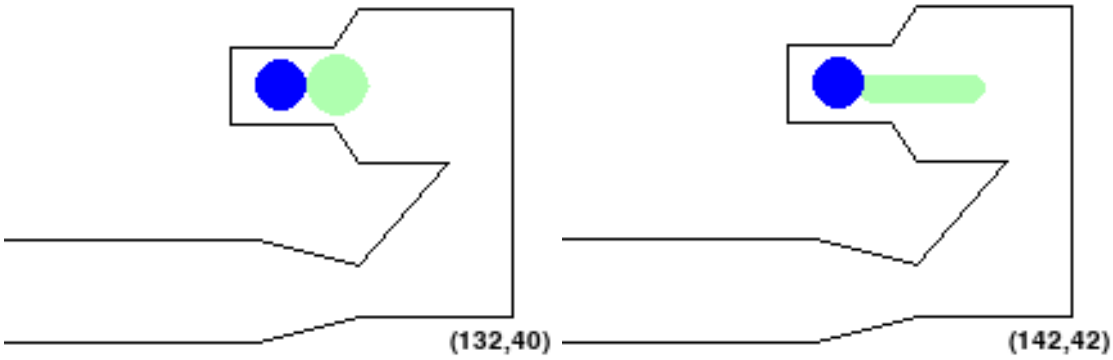


Also note that, since we are discretizing the state space, we want to add a safety buffer equal to the half diagonal of a square whose side length is the granularity we're using to digitize our map. Specifically, if the granularity is G, we want our alien to be at least $\frac{G}{\sqrt{2}}$ units way from any obstacles and the same distance away from the edges of the workspace. This ensures that the alien will not accidentally bump into an obstacle of go out of bounds while executing its path in the real world. As a consequence, configurations like this one, where the granularity is set to 10 pixels:



Note that it seems that the alien turned red despite not visibly colliding with any wall. That is because at this coarse granularity, the safety buffer is very large. Keep that in mind as you visually debug later (this effect should be less perceptible at finer granularities).

We consider the maze solved once any part of the alien's body has contacted the goal position (i.e. if any part of any of the objectives is within the alien area), as long as the alien is not violating any constraint (i.e., not out of bounds or touching a wall). Here are some valid "solved" states:



Note that for reaching the goal, we **DO NOT** apply the collision buffer we use for obstacles and out-of-bounds checking, as we want to guarantee that we actually touch the goal.

To solve this MP2, you will go through three main steps:

- Adapt your MP1 BFS code to be able to run in three dimensions.
  - This is primarily a housekeeping activity to ensure your code is not hardcoded to work for a specific type of search, but can handle generic search spaces.
- Compute a configuration space (Maze) from the original alien motion space. This configuration space is 3D because the alien robot has three degrees of freedom (x,y centroid position and shape).
- Convert that configuration space into a 3D version of MP1's map format and use your previously written 3D BFS code to compute the shortest path in this configuration space map.

In order to represent the alien's configuration space as a 3D maze, you will digitize the alien's current shape and 2D movement patterns. Each shape of the alien ('Vertical', 'Ball', 'Horizontal') can be represented by a "level" in the maze, while the walls in the maze are determined by the configurations where the current shape collides with obstacles. **It is important to remember that the robot's configuration is comprised of both its (x,y) centroid position and its shape!** In the mazes we want you to solve, the robot will have to switch its shape to reach the goal.

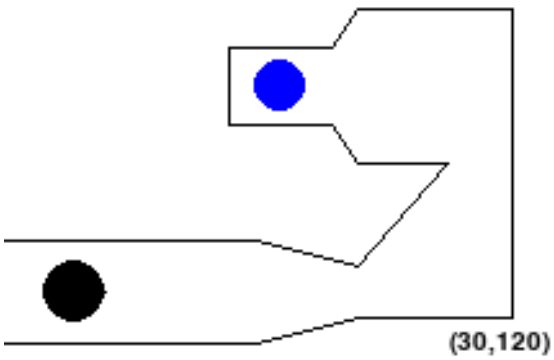
Part 0: Understanding Map configuration

Each scene (or problem) is defined in a settings file. The settings file must specify the alien's inital position, geometry of the alien in its different configurations, the goals and their geometry and the edges of the workspace. Here is a sample scene configuration:

```
[Test1]
Window : (300, 200)                    # (Width, Height)
Obstacles : [
                (0,100,100,100),   #(startx,starty,endx,endy)
                (0,140,100,140),
                (100,100,140,110),
                (100,140,140,130),
                (140,110,175,70),
                (140,130,200,130),
                (200,130,200,10),
                (200,10,140,10),
                (175,70,140,70),
                (140,70,130,55),
                (140,10,130,25),
                (130,55,90,55),
                (130,25,90,25),
                (90,55,90,25)
            ]
Goals : [
            (110, 40, 10)              # (x-coordinate, y-coordinate, radius)
        ]
Lengths: [40,0,40]
Widths: [11,25,11]
StartPoint: [30,120]
```

- Window: The window size for the given example map is 300x200 pixels. **Note that (0, 0) is the top-left corner and (300, 200) is the bottom-right corner** .
- Lengths: The lengths of the robot are in the form ['Horizontal Length', 'Ball length (always 0)', 'Vertical Length']. The length of the robot is defined by the distance between its head and tail
- Widths: widths of the robot represent the radius of the circle that is added to the line segment "body" of the alien, i.e.. how far away from the line segment defining the body is still considered to be "inside" the robot. These are ordered in the same manner as the Lengths
- Obstacles: There are many walls in the maze which are represented by a list of endpoints for line segments in the format (startx, starty, endx, endy)
- A list of disk goals specified in the form (x,y,radius). In this particular maze, there is one goal at (110,40) that has radius 10 pixels.
- The alien is set to start at the position (30,120), in its default disk configuration
- The name of this map is Test1

Here is how the map from this configuration looks:



You can play with the maps (stored in config file "maps/test_config.txt") by manually controlling the robot by executing the follwing command:

python3 mp2.py --human --map Test1 --config maps/test_config.txt

Feel free to modify the config files to do more self tests by adding test maps of your own.

Once the window pops up, you can manipulate the alien using the following keys:

- w / a / s / d: move the alien up / left / down/ right
- q / e: switch between horizontal, ball, and vertical forms of the alien. "q" will cycle backwards (Vertical -> Ball -> Horizontal) and "e" will cycle forwards (Horizontal -> Ball -> Vertical

While implementing your geometry.py file, you can also use this mode to manually verify your solution, as the alien should turn red when touching an obstacle or out of bounds, and should turn green when validly completing the course, as shown in the initial figures.

Part 1: Adapt BFS

The first part of this MP is to extend your BFS code from MP1 to handle 3D mazes. Your BFS code must be able to handle the possibility that there are multiple goals. However, unlike MP 1, your code does not have to touch all the goals. It should consider the maze "solved" when it reaches the first goal. Your new BFS code should be added to this file:

search.py

- bfs(maze): returns optimal path in a list, which contains start and objectives. If no path found, return None.

Run part 1 by using the following command

python3 part1.py mazes/small-3d

You can control the agent with keyboard inputs as usual by using arrow keys to navigate on the same level and 'u' and 'd' to move up/down a level

python3 part1.py mazes/small-3d --human

NOTE: unlike MP1, in this MP the maze might not have a path! So be sure to handle this case properly and return None! Also notice that the bfs function now takes an extra argument `ispart1`. You should pass this argument as it is when you call `getNeighbors()` Don't change it as it will mess up the tests.

Part 2: Geometry

The second part of this MP is to work out the geometrical details to build your configuration space map.

The alien robot has two forms that are represented with geometric shapes:

Form 1 (Ball): A disk with a fixed radius. This entire disk is the alien's body.

Form 2 (Oblong): An oblong (sausage shape). This is represented as a line segment with a fixed length and a radius. Any point within the specified radius of the line segment belongs to the alien's body, hence its sausage-like appearance.

We provide a helper class to help you: Alien. **Do not modify this class** . To solve MP2, you will likely find the following Alien methods to be useful:

alien.py

- get_centroid(): Returns the centroid position of the alien (x,y)
- get_head_and_tail(): Returns a list with the (x,y) coordinates of the alien's head and tail [(x_head,y_head), (x_tail,y_tail)], which are coincidental if the alien is in its disk form.
- get_length(): Returns the length of the line segment of the current alien shape
- get_width(): Returns the radius of the current shape. In the ball form this is just simply the radius. In the oblong form, this is half of the width of the oblong, which defines the distance "d" for the sausage shape.
- is_circle(): Returns whether the alien is in circle or oblong form. True if alien is in circle form, False if oblong form.
- set_alien_pos(pos): Sets the alien's centroid position to the specified pos argument
- set_alien_shape(shape): transforms the alien to the passed shape
- set_alien_configuration((x,y,shape)): Places the alien in that specified configuration, placing its centroid in position (x,y) and forcing its shape to be "shape"

The main geometry problem is then to check whether

- the alien touches the goal.
- the alien runs into obstacles
- all parts of the alien remain within the given window (bounds)

To do this, you need to implement the following functions:

geometry.py

- does_alien_touch_walls(alien, walls, granularity): Determine whether any part of the alien touches the walls given a certain space granularity, returns True if it does and False otherwise.
- does_alien_touch_goals(alien, goals): Determine whether any part of the alien touches goals, returns True if it does and False otherwise.
- is_alien_within_window(alien, window, granularity):Determine whether the alien stays within the window given a certain space granularity, returns True if it does and False otherwise.

It will be helpful to look into computing distances between a point and a line segment for these functions, as well as computing distances between two line segments.

Remember again that we want our alien to be at least $\frac{Granularity}{\sqrt{2}}$ away from any obstacles and from being out of bounds. **Also note that. when checking for equalities using floating point numbers, you can get inconsistent values for the least significant bits due to machine precision. Using np.isclose(x,y) IS NECESSARY TO ENSURE CONSISTENCY in your results**

In addition, in robotics, it is commonplace to err on the side of caution - Therefore, **If the alien is found to be TANGENT to either the WALLS or the BOUNDARIES it should be considered as AN INVALID configuration - i.e. should return True in the collision checking.** We have built in some assertions at the bottom of geometry.py to help with basic debugging, which can be executed by:

python3 geometry.py

As mentioned before, once this class is properly implemented, you can also perform visual validation by running:

python3 mp2.py --human --map [MapName] --config maps/test_config.txt

The file maps/test_config.txt contains several maps. MapName is the name of the one you'd like to run. That is, MapName should be Test1, Test2, Test3, Test4, or NoSolutionMap.

Part 3: Transformation to Maze

The third part of this MP is to transform the movement and transformation space of the alien into a 3D maze. As mentioned before, the alien has three degrees of freedom: x,y, and shape. You can find more details about these shapes in **alien.py**

In each shape, the alien's centroid position is limited to a different movement space due to new constraints imposed by the new shape. This is the key to this part of the MP.

Your main task for part 3 of this MP is to use the geometry functions defined in the previous section to construct an MP1-style maze to be transversed by the alien during its motion. This maze will have three levels, one for each body shape. Specifically:

- 'Horizontal' is level 0
- 'Ball' is level 1
- 'Vertical' is level 2

Within each level, the rows and columns in the maze match to the robot's (x,y) centroid position. The number of rows and columns of the maze is defined by the dimensions of the workspace as well as the chosen granularity. The number of positions in the row or column is $\frac{\text{maxposition}-\text{minposition}}{\text{granularity}}+1$. For this MP, minposition is always zero.

In order to ensure uniformity in the shapes of the mazes, we provide you with two helper functions in util.py, configToIdx and idxToConfig. These functions will give you the correct correspondence between a robot configuration and a maze index. **YOU MUST USE THEM TO ENSURE CONSISTENCY FOR THE AUTOGRADER** and to make your lives easier.

In order to perform this task, you will define the function **transformToMaze(alien,goals,walls,window, granularity)** in **transform.py**

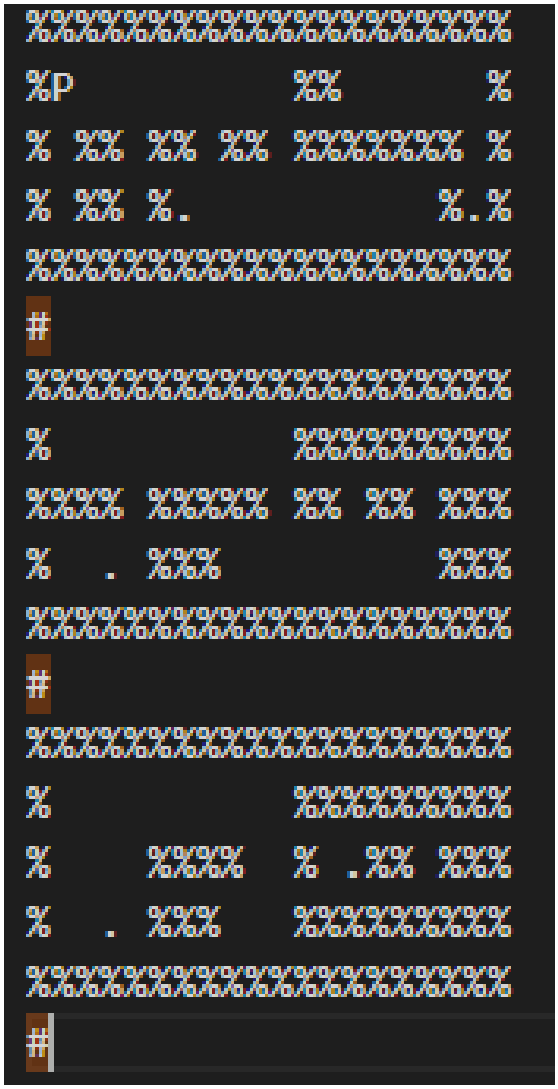
- transformToMaze(alien, goals, walls, window,granularity): This function takes in an alien instance, list of goals, list of line segments representing walls, and a window size and a granularity. It then returns a maze instance based on these arguments.
- The granularity parameter determines how many cells will be contained in a given interval.
- The maze class can be found in **maze.py**.
- We then generate a maze instance by calling the maze constructor from **maze.py** and passing in the path to the ASCII file.

The maze file is defined by an ASCII file with the following conventions:

- **wall: %**
- **start: P**
- **waypoint: .**
- **level separator/end of file marker: #**

Your generated maze must abide by the ASCII conventions above before you can instantiate a Maze object and save it to a file by calling maze.saveToFile(FILENAME).

Here is an example of a maze representation from **small-3d**:



To fill in the maze positions for each level, you will use the geometry functions you implemented in **part 2** to evaluate collision, goal, and out of bound states for centroid positions. This will define your movement space for each level. The bruteforce solution is to check all centroid positions for collisions with different maze options. However, there may be more efficient ways to do this using shortcuts to bypass unnecessary checking. We have provided you with the ground truth mazes for all the test maps calculated at 2,5 and 10 pixels of granularity. If you wish to validate your solution, you can run

python3 transform.py

While initially debugging, you might wish to change the list of granularities and maps you are building the maps for in the __main__ section of transform.py so you can get quicker feedback. By default, it will generate the maps for all

example maps at 2,5 and 10 pixels of granularity and compare those to the ground truth mazes. If it identifies discrepancies between your map and the ground truth, it will highlight those as shown below:

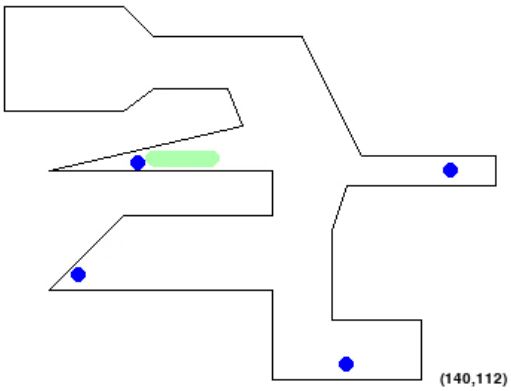
```
Differences in Test1 at granularity 2:
Ground Truth walls mistakenly identified as goals: [(26, 144, 'Vertical'), (54, 100, 'Ball'), (54, 102, 'Ball'), (74, 206, 'Horizontal')]

Ground Truth walls mistakenly identified as free space: [(8, 210, 'Vertical'), (26, 142, 'Horizontal')]

Ground Truth goals mistakenly identified as walls: [(38, 124, 'Horizontal'), (38, 126, 'Horizontal')]

Ground Truth goals mistakenly identified as free space: [(38, 130, 'Horizontal'), (38, 132, 'Horizontal'), (40, 120, 'Ball')]

Ground Truth free space mistakenly identified as walls: [(74, 96, 'Ball')]

Ground Truth free space mistakenly identified as goals: [(80, 96, 'Ball')]
```

Part 4: Searching the path in Maze

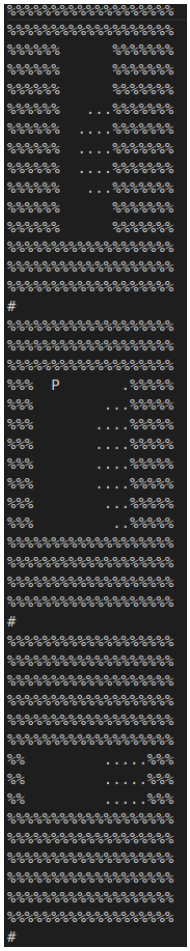
In this section, you put all of the ingredients together! You will receive a given map, discretize it using your Part 3 code, and use your BFS search algorithm to find the shortest path to any of the goals. You can test all of the components together with:

python3 mp2.py --map [MapName] --config maps/test_config.txt --save-maze [MazeDestinaton].txt

Where MapName is in [Test1,Test2,Test3,Test4,NoSolutionMap]. If all your parts are implemented correctly, you should see two outputs. The first of them is an animation of the solution you just found playing in the main screen (you can press escape to kill it).



The second of them is the maze version of the map you just solved, which can be found in the file you specified as MazeDestination. Below is an example of the maze generated for Test1 with granularity 6. Notice that these digitized mazes can get large if you select a finer granularity or a map that's more complex.



Provided Code Skeleton and Deliverables

The code you need to get started is in this zip file. You will only have to modify and submit following files:

- geometry.py
- transform.py
- search.py

Do not modify other provided code. It will make your code not runnable.

You can get additional help on the parameters for the main MP program by typing **python3 mp2.py -h** into your terminal.

Please upload **search.py, transform.py,** and **geometry.py** (all together) to gradescope.

Do not submit extra files to gradescope.