Dept. of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur

ALGORITHMS
(EC31205)

Coding Task: 1

Title: Robot Navigation Problem

Date of Submission: September 29th, 2023

Name: Irsh Vijay
Roll Number: 21EC30025

Instructors:
Prof. Debashis Sen

I. Naive Robot Navigation Problem:

Problem Statement:

The Naive Robot Navigation Problem involves designing a program to navigate a grid maze using a Breadth-First Search (BFS) algorithm. The robot's objective is to find the shortest navigation route while following only white tiles. The program takes into account various input parameters, including the maze configuration, starting and ending coordinates, and the option to allow or disallow diagonal movements.

The Code:

Allowed_directions is appended to include diagonal movements if all_direction is True. Once an allowed point is visited it is added to the set and BFS is implemented until it reaches the end.

```
def naive_robot_navigation(maze, start, end, all_direction):
  allowed_directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
  if(all_direction):
    allowed_directions.extend([(1, 1), (-1, 1), (1, -1), (-1, -1)])

  # allowed_directions contain possible move sets from a particular point

  visited = set()
  queue = [(start, [])]

  while queue:
    (row, col), path = queue.pop(0)

    if (row, col) == end:
      plot_maze(maze, start, end, path + [(row, col)])
      return True, path + [(row, col)]

    # Breadth First Search Implementation

    for dr, dc in allowed_directions:
      new_row, new_col = row + dr, col + dc

      if check_valid_move(maze, new_row, new_col, all_direction) and (new_row, new_col) not in visited:
        visited.add((new_row, new_col))
        queue.append(((new_row, new_col), path + [(row, col)]))

  plot_maze(maze, start, end)
  return False, []
```

Here the check_valid_move function is defined as:

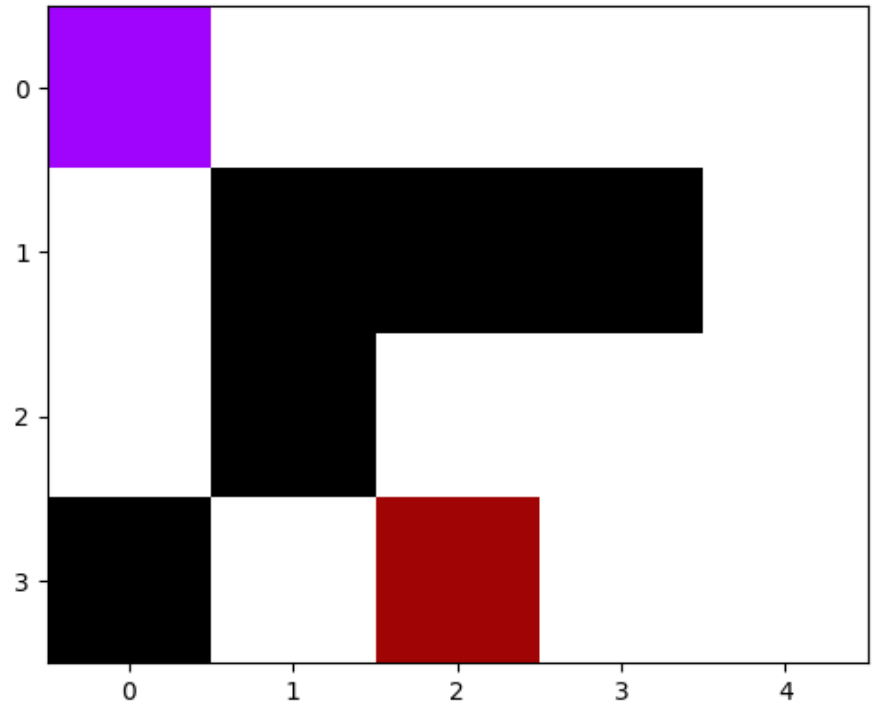
```
def check_valid_move(maze, row, col, all_direction):
  return 0 <= row < len(maze) and 0 <= col < len(maze[0]) and maze[row][col] == 1
```

Additionally, I created a plot_maze function to easily visualize the path taken by the algorithm

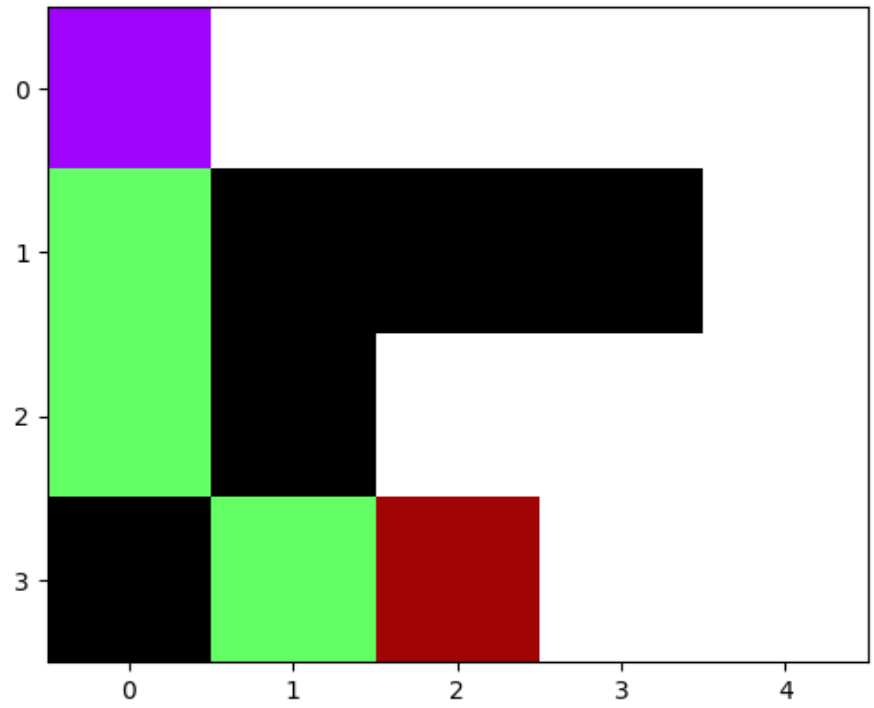
```python
import matplotlib.pyplot as plt

# A plot maze function to visualise the path taken

def plot_maze(maze, start = False, end = False, path = False):
  rows = len(maze)
  cols = len(maze[0])

  grid = [[(255, 255, 255) if maze[row][col] == 1 else (0, 0, 0) for col in range(cols)] for row in range(rows)]

  if start:
    grid[start[0]][start[1]] = (160, 4, 252)
  if end:
    grid[end[0]][end[1]] = (160, 4, 4)
  if path:
    for path_row, path_col in path:
      if(grid[path_row][path_col] != (160, 4, 252) and grid[path_row][path_col] != (160, 4, 4)):
        grid[path_row][path_col] = (100, 255, 100)

  plt.imshow(grid)
  plt.xticks(range(cols))
  plt.yticks(range(rows))
  plt.show()
```

Few Test Cases:

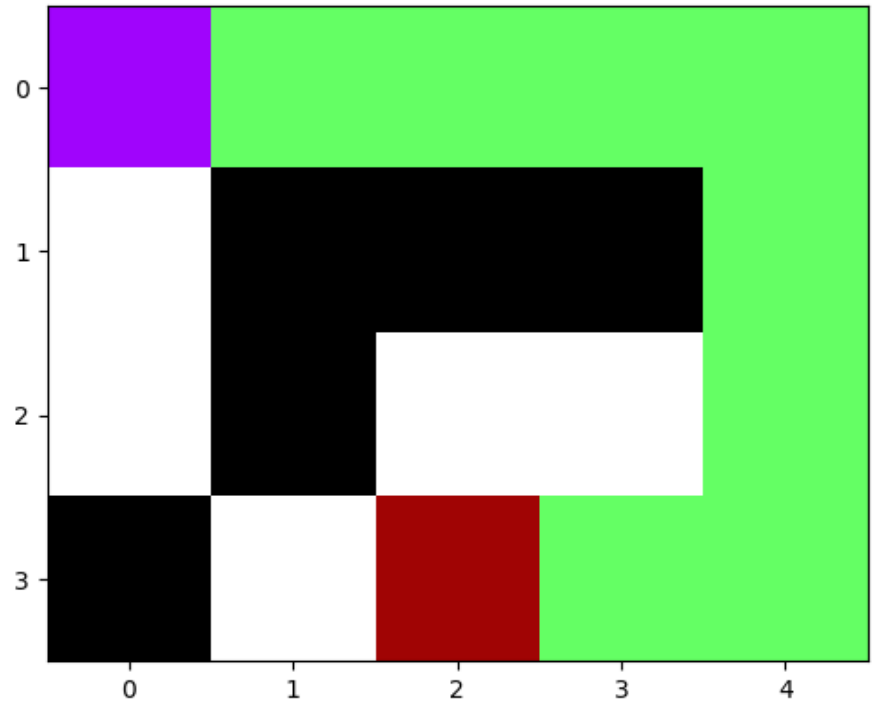
Given Maze (Purple: Start, Red: End):



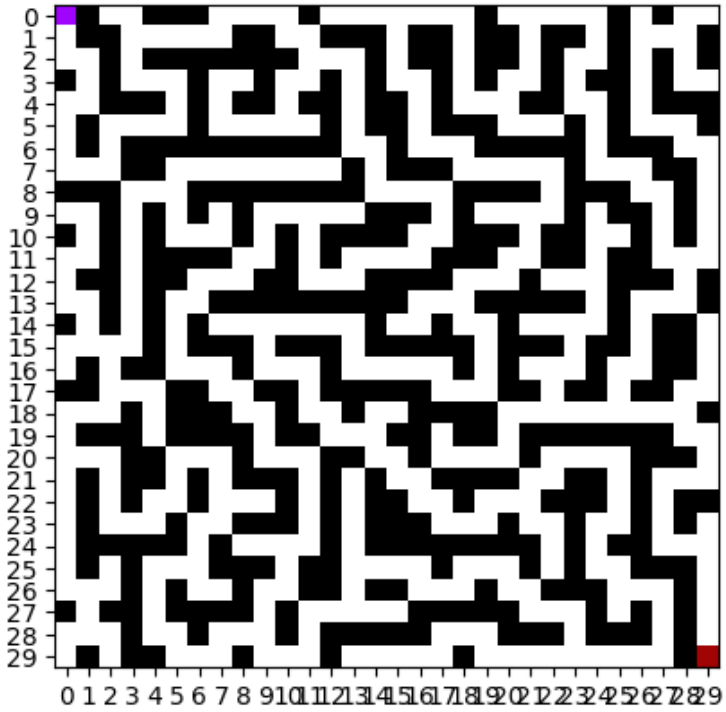
If all directions are allowed:



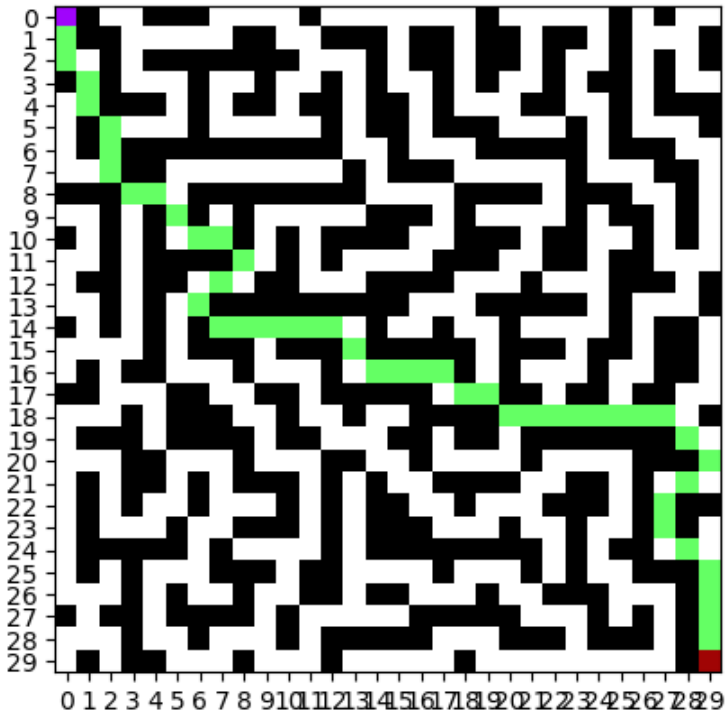
If all directions aren't allowed:



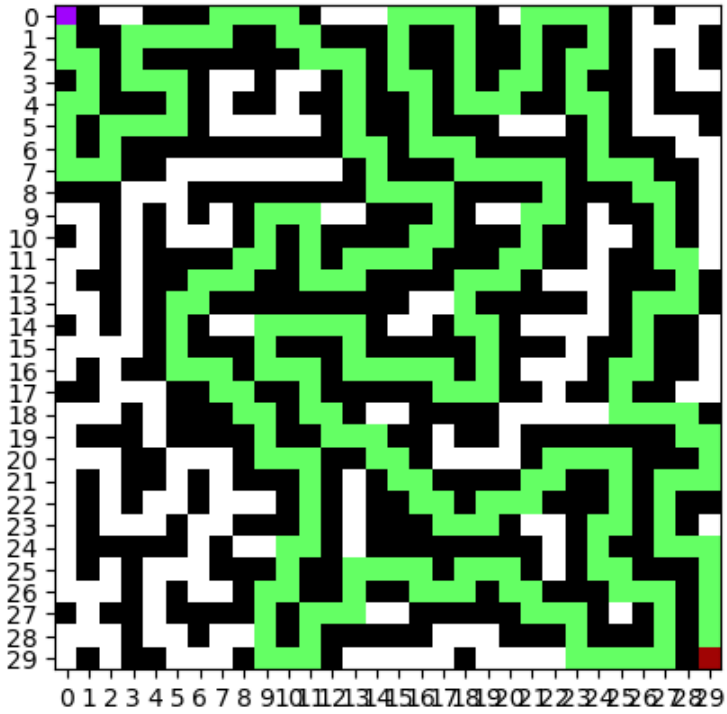
On a bit more complex maze:



All_directions are allowed:



All_directions aren’t allowed:



Results:
The program provides two outputs in the form of a tuple:

- **Reachability:** The first element of the tuple is a boolean value that indicates whether the endpoint is reachable from the starting point. It returns True if the endpoint is reachable and False if it is not.
- **Navigation Path:** The second element of the tuple is a list that represents the navigation plan for the robot. This list contains a sequence of coordinates representing the path from the starting point to the endpoint, considering the shortest possible route.

II. Smarter Robot Navigation Problem:

Problem Statement:
The Smarter Robot Navigation Problem builds upon the previous Naive Robot Navigation Problem by considering the loss of speed due to changing directions. The objective is to find the shortest navigation route for a robot navigating a grid maze while taking into account the overhead costs associated with different degrees of turns. The program should use Dijkstra's shortest path algorithm to determine the optimal route.

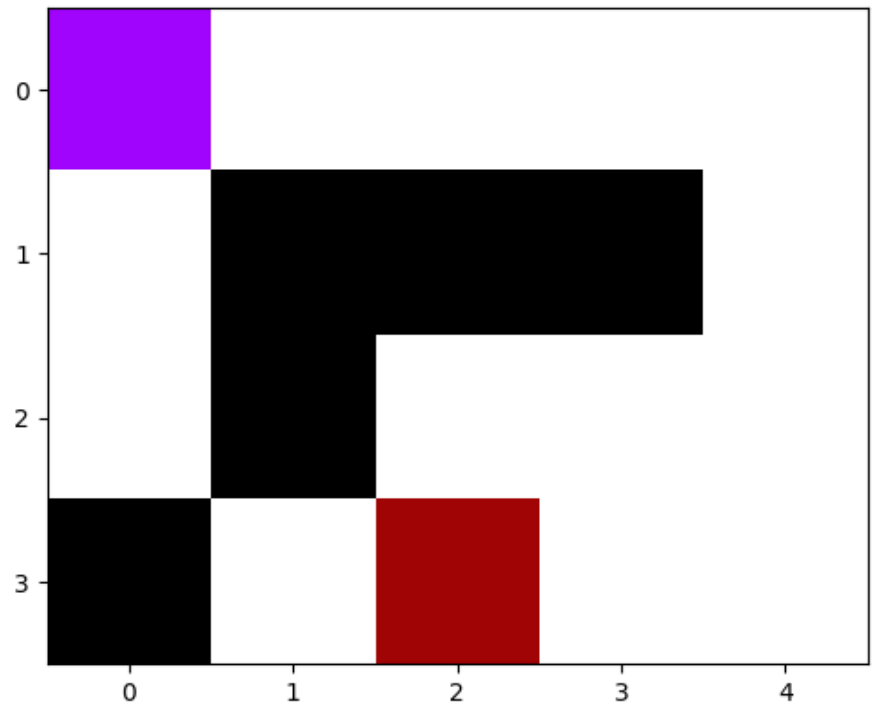
The Code:
This code uses Dijkstra’s Algorithm, an additional directional_cost function has been added, this

```python
import math

def directional_cost(prev_direction, direction):
    # Calculate the directional cost based on the change in direction
    ang_prev =  math.atan2(prev_direction[1], prev_direction[0])/math.pi*180
    ang_new =  math.atan2(direction[1], direction[0])/math.pi*180
    if ang_prev == ang_new:
        return 0.0  # No turn
    elif abs(ang_prev - ang_new)==45 or abs(ang_prev - ang_new)==135:
        return 0.5  # 45-degree turn
    else:
        return 1.0  # 90-degree turn
```

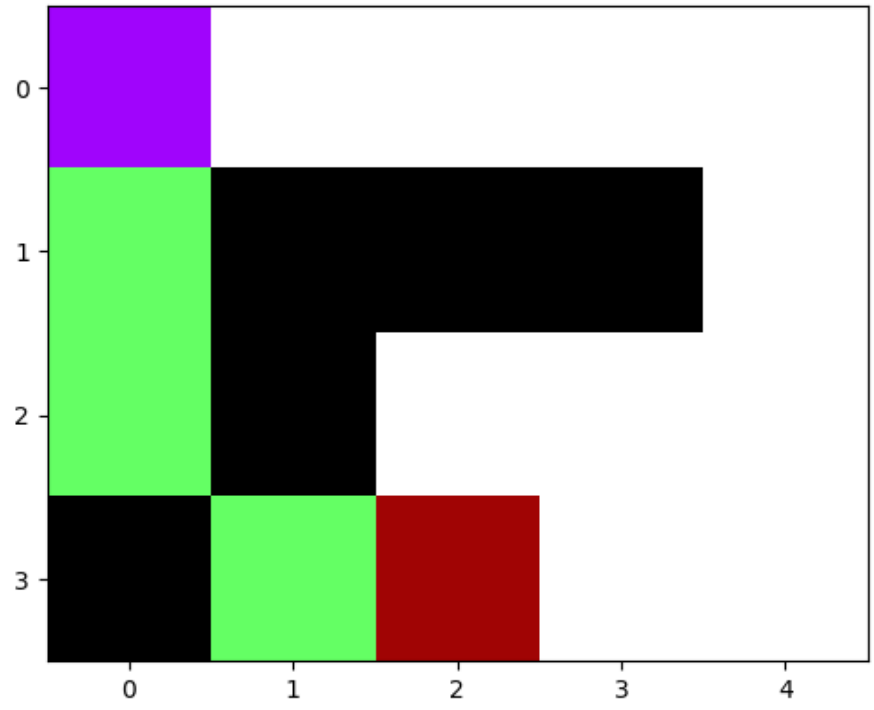
```python
import numpy as np
import heapq

def smarter_robot_navigation(maze, start, end, all_direction):
    allowed_directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    if all_direction:
        allowed_directions.extend([(1, 1), (-1, 1), (1, -1), (-1, -1)])

    visited = set()
    cost_map = {}
    queue = [(0, start, [], (None, None))]

    while queue:
        current_cost, (row, col), path, prev_direction = heapq.heappop(queue)

        if (row, col) == end:
            plot_maze(maze, start, end, path + [(row, col)])
            total_cost = calculate_total_cost(path)
            return True, path + [(row, col)], total_cost

        if (row, col) in visited:
            continue

        visited.add((row, col))

        for dr, dc in allowed_directions:
            new_row, new_col = row + dr, col + dc
            new_direction = (dr, dc)

            if check_valid_move(maze, new_row, new_col, all_direction):
                if prev_direction[0] == None:
                    step_cost = 0.0
                else:
                    step_cost = directional_cost(prev_direction, new_direction)
                new_cost = current_cost + step_cost
                heapq.heappush(queue, (new_cost, (new_row, new_col), path + [(row, col)], new_direction))
    return False, [], 'inf'
```

Few Test Cases:

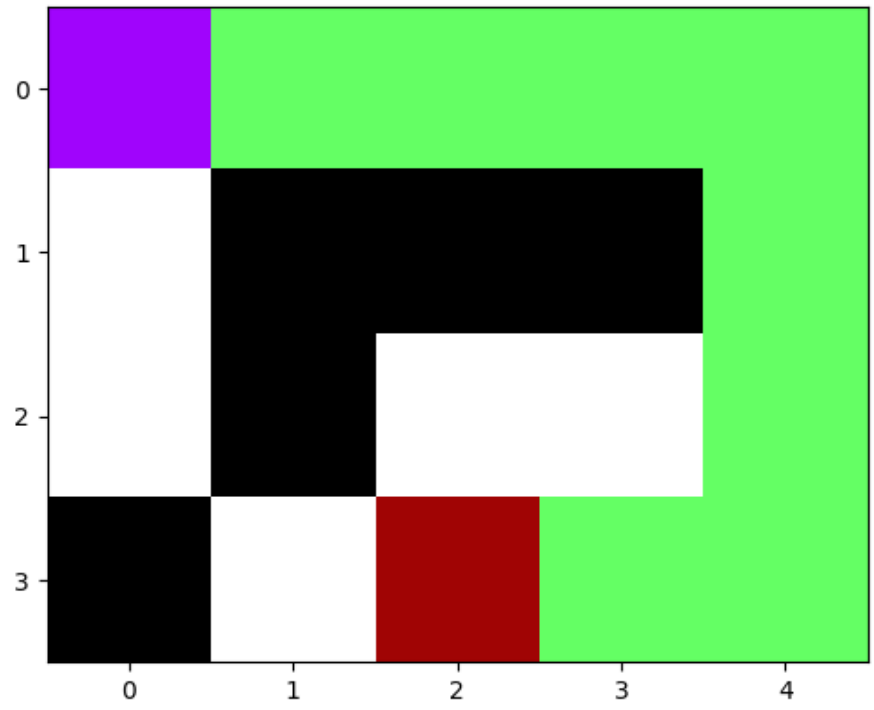
Given Maze (Purple: Start, Red: End):



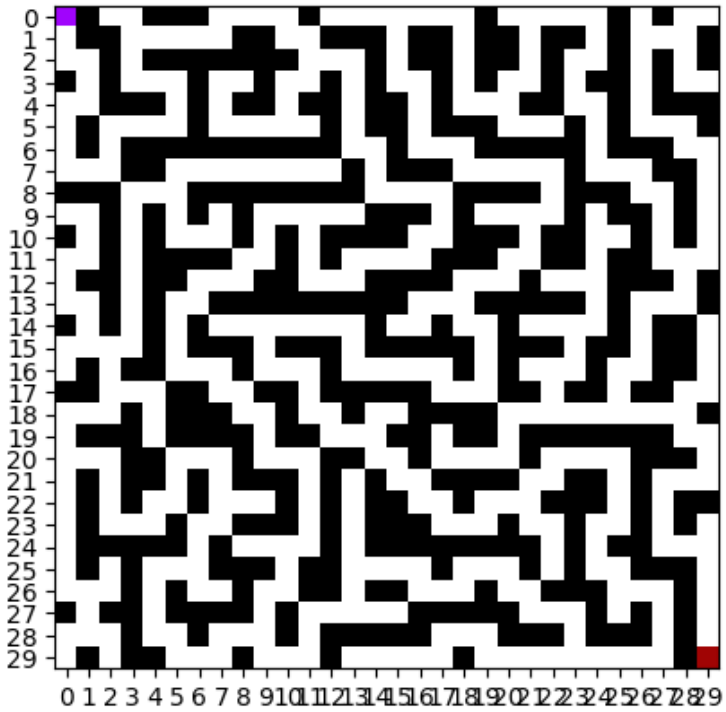
If all directions are allowed (Cost = 5.0):



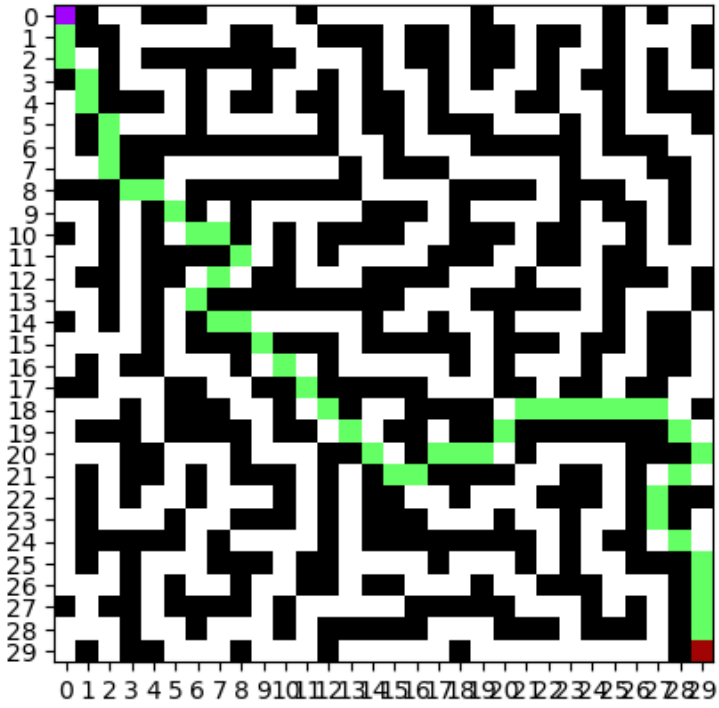
If all directions aren’t allowed (Cost = 11.0):



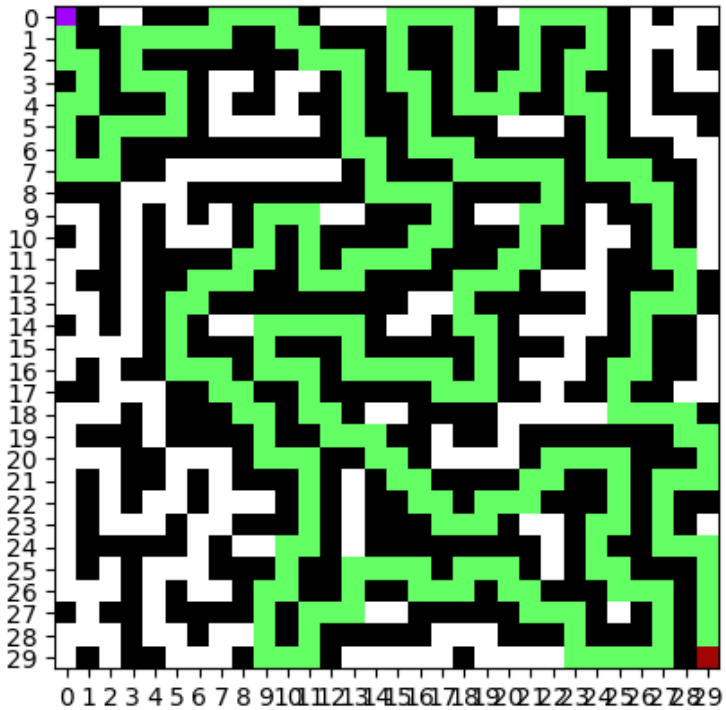
On a bit more complex maze:



All_directions are allowed (Cost=63.5):

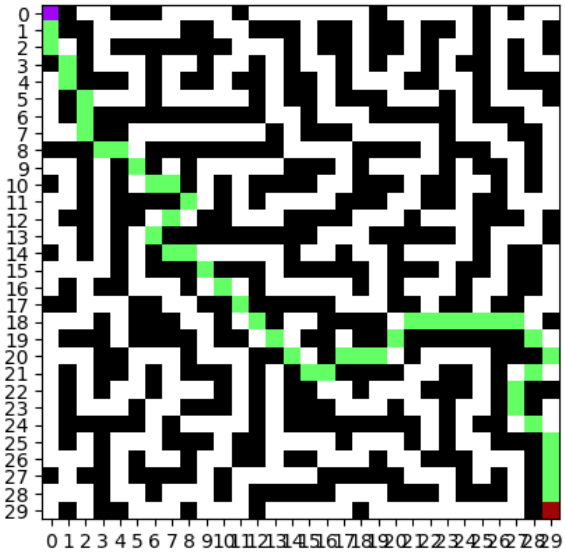


All_directions aren’t allowed (Cost = 406.0):

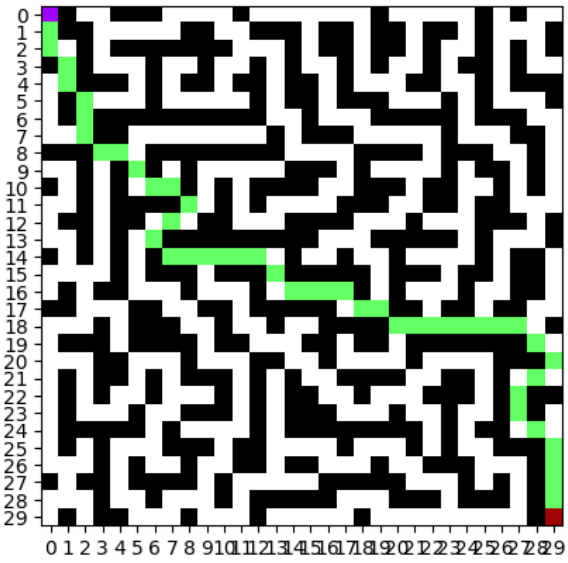


Results:

We can see a difference between outputs in the bigger maze when all_directions are allowed using the naive and smarter robot navigation:



Smarter Robot Navigation



Naive Robot Navigation