Analyzing the Difficulty of Graph Theory Problems

Internal Assessment

Subject: Mathematics HL

Session: May 2020

Author: Aaron Esau

Word Count: 1909

Contents

Analyzing the Difficulty of Graph Theory Problems

1 Introduction

Have you ever considered how Google Maps works? To be able to find the best (i.e. least time-consuming) route from a source to a destination, Google must keep track of a lot of information on every road such as speed limit, distance, and knowledge regarding adjacent roads.

The problem of finding optimized routes can be solved using graph theory– where the intersection of two or more roads is a vertex, and the roads themselves are edges. The length of each edge is simply the speed a car can travel multiplied by the length of the road.

There are hundreds, thousands, millions, or billions of possible paths between any two given starting and ending points. Thus, it would require a great amount of computational power to have a computer program calculate all paths. But, all paths from a source to a destination must be considered when determining how "difficult" a path is.

In this paper, we will analyze data relating to the numerous possible paths from a source to a destination for a given graph theory problem. And, we will identify two ways to rank the "difficulty" of graph theory problems.

1.1 Analogy of a Maze

Using an analogy such as a maze makes the problem easier to assess and tackle. A simple analogy of the real-life scenario is a maze that has multiple paths to reach the end (the solution).

1.2 Decontextualization

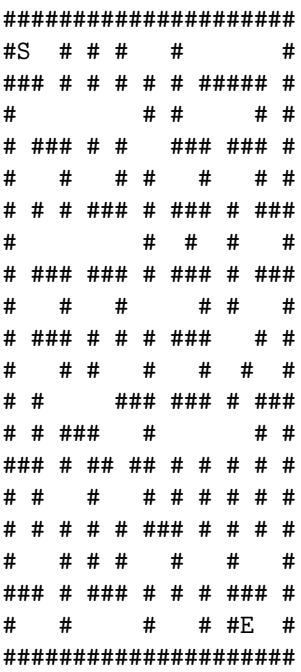

Figure 1: A maze with multiple solutions

In the maze in Figure 1, the space character represents a space the solver may travel through, while the # character represents walls. The character S represents the start of the maze while the E represents the end. The "intersection" of "channels" the solver may travel in represent vertices, and the channels themselves are the edges. The length of every channel represents the difficulty of taking that path.

Every space that the solver may travel through called a node. Hence, every vertice is a node, and edges may consist of multiple nodes. A path is simply the route of vertices and edges to follow from the start to the end. The difficulty of any given path is simply the count of the number of nodes in that path. In the context of driving route-finding, the difficulty is the time to reach a destination by car.

Of course, there are multiple possible solutions to the maze in Figure 1 (and other mazes in the analogy) that have varying difficulty.

Additionally, it is important to understand that the "difficulty" to reach a destination by car is represented by the "overall" difficulty of the maze.

2 Analysis

2.1 Obtaining the Data

I wrote a program (the Python 3 code is listed in Appendix 5.1) that first reads a file named *maze.txt* which contains the maze to solve.

2.1.1 High-Level Overview

The program then finds every possible path from S to E in the maze. For every solution path of the maze, the program calculates the difficulty (i.e. number of movements required to reach the end) and stores it in a separate list.

As the program has a list of difficulties for every solution in the maze, it now sorts the list in reverse order (i.e. descending–from highest to lowest difficulty).

2.1.2 Path-Finding Algorithm

The program uses a modified A* search algorithm to traverse the graph. A quick summary of the algorithm is:

1. For every potential node extension from the current node:
    (a) If the node has been explored already, continue to the next potential node.
    (b) If the current node is the end node, add the current path to a list of solution paths and continue to the next potential node.
    (c) Mark the current node being tested as explored.
    (d) Recurse by repeating the same procedure on the current node being tested.

Although the algorithm is not the focus of this paper, it is important to understand that the order which the algorithm discovers paths in is arbitrary; that is, there is no way to predict if the first path discovered will be a relatively easy or difficult solution. Thus, to create a graph of the path difficulties that has a useful trend, it is necessary to first sort the path difficulties in some way. Additionally, to find *all* optimal paths to the end, when using any algorithm, all paths must be discovered.

2.2 Curve Fitting

The list of paths is used as the y value set, and corresponding, sequential, integer x values (the "index" of the path) are assigned starting at 1.

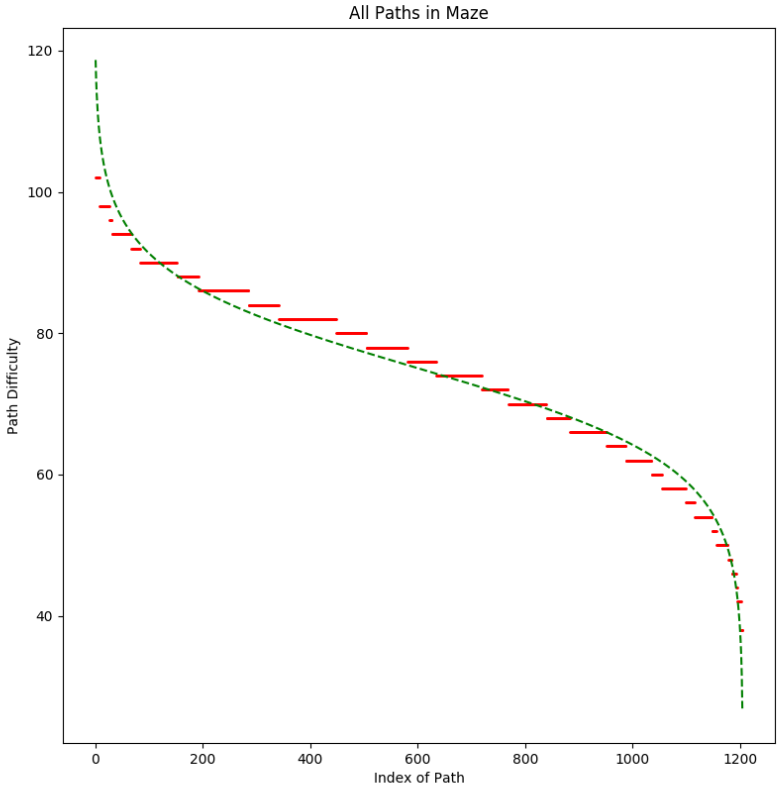

Figure 2: Reverse-ordered path difficulties graphed with a curve of best fit

So, in the graph shown in Figure 2, the y axis is the difficulty of each path. For every solution in the maze, there is a unique integer x-value; the x values are discrete. For instance, the x-value 200 in Figure 2 a single, unique path in the maze which has a difficulty of 86.

The graph clearly resembles a reverse sigmoid-type function. A sigmoid function makes sense in context because the values are limited to a range of values; there should be asymptotes at 0 and the x-value of the maximum number of solutions for a given maze. The sigmoid function is only reversed (*f(-x)*) because the program ordered the list in reverse (descending path difficulties).

Finally, as the graph of the reverse-ordered list of path difficulties for the maze in Figure 1 appears to be a sigmoid-type function, using the Least Squares method, some curve-fitting code in the program automatically identifies the "best" values (i.e. values which produce a standard error closest to 0) for y_0 and k in the function:

$$f(x) = \frac{\ln\left(\frac{-L}{(x-L)} - 1\right)}{-k} + y_0$$

where:

L is the maximum x-value of the function

y_0 is the sigmoid's midpoint y-value

k is the curve's growth rate

Figure 3: Best-fit sigmoid curve function for population data

2.2.1 Best-Fit Values

In Figure 2, the values of the constants for the sigmoid curve (automatically determined by the program) of best-fit function are:

$L = 1204$ *(maximum x-value)*

$y_0 \approx 75.0$ *(midpoint y-value)*

$k \approx 0.147$ *(growth rate)*

An explanation of all of the above values will be provided in the following three sections.

For the above constants whose values require optimization by the program to determine, the standard error values (in the context of the maze in Figure 1's data) are:

$\sigma_{y_0} \approx 0.0607$

$\sigma_k \approx 0.000731$

2.2.2 Understanding the Constant L

How does the value impact the graph? The constant L simply defines an asymptote at the desired x-value.

Where does the value come from? The maximum x-value is known because the x values are never taken from a sample (rather, they are generated by the program); hence, the L value that should be used in the function in Figure 3 is always simply the largest number in the set of indexes. There is no need to perform optimization to determine the best L value.

What does the value mean in context of the maze? Because the function should model the data in the sample, it does not make sense to set a value for L other than the maximum x-value in the set of indexes.

2.2.3 Understanding the Constant y_0

How does the value impact the graph? The constant y_0 shifts the graph "upward" as it is simply added at the end of the function.

Where does the value come from? Although the y_0 value should be approximately half of the maximum y-value in the sample of path difficulties as y_0 is the midpoint, this is not always the case. For example, more difficult mazes

most likely have a steeper curve on the left half of the graph as there are more, more difficult paths to the solution. Likewise, in a less difficult maze, there are likely more, less difficult paths.

What does the value mean in context of the maze? Because of this feature, one potential measure of difficulty of the maze is to compare the first and last path difficulty values. This idea will be explored in Section 3.1.

2.2.4 Understanding the Constant k

How does the value impact the graph? Changing the constant k changes the slope and shape of the curve.

Where does the value come from? As the change is reflected on both halves of the graph (i.e. quadrants II and III with I and IV) there is little information that can be directly based on the value of k that is useful in determining difficulty.

What does the value mean in context of the maze? Although, there is information that is indirectly based on the value of k through the integration of the curve. This idea will be explored in Section 3.2.

2.3 Sampling from the Data

The goal is to model paths from a source to destination, regardless of if the number of potential paths is extremely large. There may simply not enough computing power to analyze larger mazes. Thus, it is necessary to have the ability to analyze samples rather than the whole population of data.

As discussed in Section 2.1.2, before sorting the set of path difficulties, the order is entirely arbitrary. Thus, not all solutions to the maze are required to observe the trend. However, the more data obtained, the more the trend of the sample data will match that of the population data.

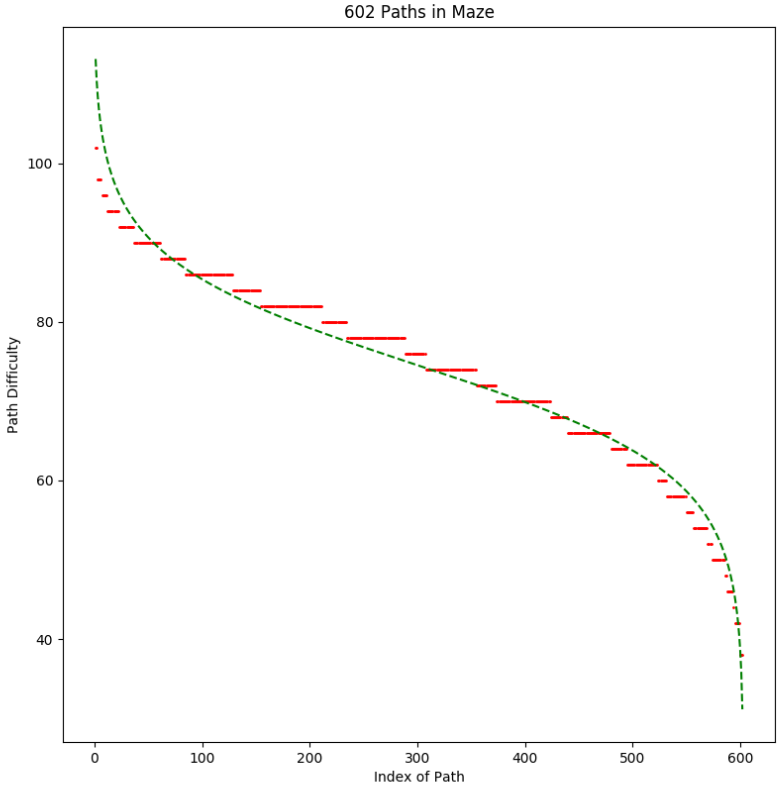

Figure 4: Best-fit curve for sample of data

For example, whereas the population size of the maze in Figure 1 graphed in Figure 2 is 1204, the same trend can be observed in Figure 4 when only identifying 602 paths–half as many as that of the population. To generate this data, I simply configured the program to stop after finding 602 (an arbitrary number) paths.

3 Determining the Overall Difficulty of Mazes

A measure of overall difficulty of a maze must be based on the difficulty of every individual path as there are, inherently, multiple possible solutions.

Note that, for both methods outlined in Section 3, the data used for com-

parisons will be the population data (n=1204) from the maze in Figure 1. The best-fit function and constants used will be from Figure 3. However, in a real-life scenario, a sample of the data would be used.

3.1 "Head to Tail" Ratio

3.1.1 Explanation

A more difficult maze will have relatively more difficult paths. Likewise, a less difficult maze will have relatively more, less difficult paths.

So, a possible measurement for the overall difficulty of a maze is the ratio of the difficulty of the most difficult to the least difficult path.

3.1.2 Calculations

$$ratio = \frac{\max}{\min} = \frac{102}{38} \approx 2.68$$

Figure 5: Head-tail measurement of overall difficulty

Hence, the score of the maze is 2.68.

3.1.3 Disadvantages

Although quick and computationally cheap, this method is not fully representative of the maze as only the most and least difficult paths are used in the calculation of the measurement.

3.2 Ratio of Integrals

3.2.1 Explanation

It is more effective to consider the whole maze as opposed to only the maximum and minimum values for path difficulties by comparing the area of the

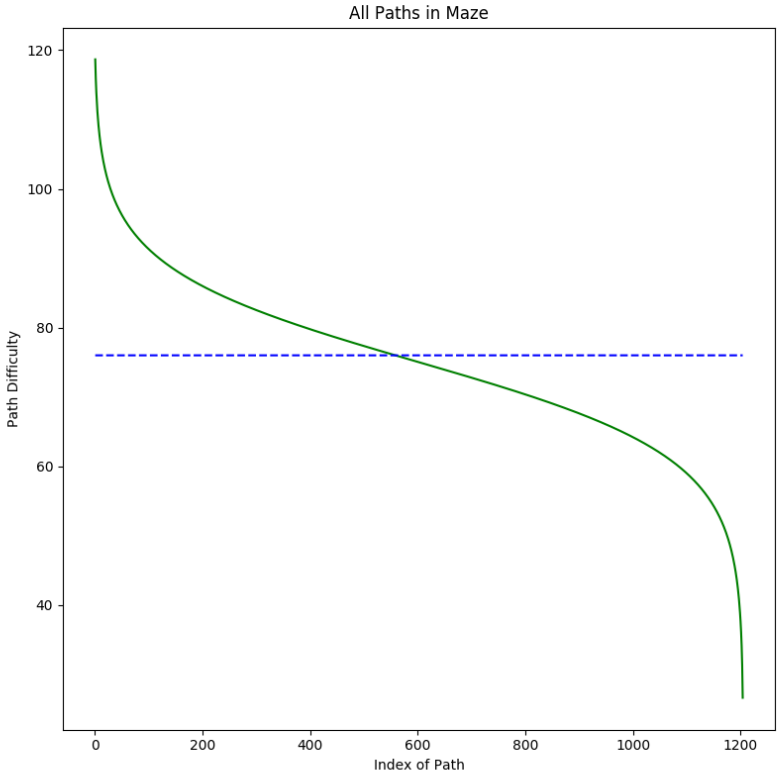

Figure 6: Curve and median line

harder half (quadrants II and III) of the paths to that of the easier half (quadrants I and IV).

To use this method, we simply integrate the leftmost half of the curve and compare the ratio to rightmost (both relative to a dividing line, as defined in the next section).

3.2.2 Calculations

The dividing line to integrate against is $y=a$ where a is the median path difficulty, as shown in Figure 6.

$$
\begin{aligned}
ratio &= \frac{\int_0^{\frac{\widetilde{x}}{2}} f\left(x\right)\ dx}{\int_{\frac{\widetilde{x}}{2}}^{\widetilde{x}} f\left(x\right)\ dx} \\
&= \frac{\left(\int_0^{\frac{\widetilde{x}}{2}} \left(\frac{\ln\left(\frac{-L}{(x-L)}-1\right)}{-k}+y_0\right) dx\right)}{\left(\int_{\frac{\widetilde{x}}{2}}^{\widetilde{x}} \left(\frac{\ln\left(\frac{-L}{(x-L)}-1\right)}{-k}+y_0\right) dx\right)} \\
&= \frac{\left[\frac{-(x-L)\ln\left(\left|\frac{x}{x-L}\right|\right)-L\ln(|x|)}{k}+y_0x\right]_0^{\frac{\widetilde{x}}{2}}}{\left[\frac{-(x-L)\ln\left(\left|\frac{x}{x-L}\right|\right)-L\ln(|x|)}{k}+y_0x\right]_{\frac{\widetilde{x}}{2}}^{\widetilde{x}}}
\end{aligned}
$$

Hence, calculate the ratio score for the maze in Figure 1 using the values:

$\widetilde{x}\ = 76$ *(median path difficulty)*

$L\ = 1204$ *(sample size)*

$y_0\ \approx 75.0$ *(from optimization)*

$k\ \approx 0.147$ *(from optimization)*

$$
\begin{aligned}
ratio &\approx \frac{4001.67999653}{3633.98663978} \\
\implies \qquad &\approx 1.10
\end{aligned}
$$

The higher the ratio, the more difficult the maze is. A ratio less than 1 indicates that the area of the easier paths is greater than that of the harder paths relative to the median path.

4 Conclusion

By simplifying a pathfinding problem down to a multiple-solution maze, we discovered that solutions found using graph theory, when ordered and graphed, have a sigmoid shape. We observed that a trend begins to appear as more solutions are collected. Moreover, we analyzed two potential methods to score how

difficult graph theory problems are to solve–neither of which rely on knowing every solution to problems.

The multi-solution maze in Figure 1 was arbitrarily created by me to use as an example. However, the program in Appendix 5.1 works for all mazes.

Possible extensions of this work include investigating:

- How does adding or removing a "wall" in a maze impact its difficulty score?
- What is the maximum possible head-tail ratio difficulty score? What would such a maze look like? And, what would a graph theory problem with a higher difficulty ratio look like?
- How does the sigmoid trend begin to appear as we collect more data? Is there an algorithm we could use to estimate the sigmoid trend while collecting fewer solution path data points?

5 Appendix

5.1 solver.py

```
#!/usr/bin/env python3

from scipy.optimize import curve_fit
from numpy import array, tan, arctan, diag, sqrt, log,
    percentile, e
from matplotlib import pyplot as plt



# load in a maze line by line
def load_maze(maze):
    return [list(x) for x in maze.strip().split('\n')]



def explore_node(maze, position, explored_paths = list())
    :
    paths_to_end = list()

    # where can we explore?
    for change in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        # see where we'd be if we applied the change
        test_position = (position[0] + change[0],
            position[1] + change[1])

        if test_position in explored_paths:
            # previously explored path
            continue
        else:
```

```
        # let's keep track of this test
        explored_paths.append(test_position)

    # check if this change is permitted
    if min(test_position[0], test_position[1]) < 0 or
        (test_position[0] >= len(maze) or
        test_position[1] >= len(maze[0])):
        # outside of the map
        continue

    character_at_test_position = maze[test_position
        [0]][test_position[1]]

    if character_at_test_position == 'E':
        # we found the end!
        paths_to_end.append([change])
        continue
    elif character_at_test_position != ' ':
        # we're only appowed to pass through a space
        continue

    # let's recurse that path
    node_paths = explore_node(maze, test_position,
        explored_paths = explored_paths.copy())

    if not node_paths:
        # we found no ways to get to end from this
            node
        continue
```

```
            # let's keep track of all paths to end we find
            for path_to_end in node_paths:
                paths_to_end.append([change] + path_to_end)

    # return all paths to the end that we found
    return paths_to_end


def print_maze(maze):
    print('\n'.join([''.join(row) for row in maze]))


def solve_maze(maze):
    paths = list()

    start_position, end_position = None, None

    # find the start and end position
    for y in range(len(maze)):
        if 'S' in maze[y]:
            start_position = (y, maze[y].index('S'))
        elif 'E' in maze[y]:
            end_position = (y, maze[y].index('E'))

    # recurse starting from initial node
    paths = explore_node(maze, start_position)

    return (start_position, paths)
```

```
if __name__ == '__main__':
    # load / parse maze file
    with open('maze.txt', 'r') as f:
        maze = load_maze(f.read())

    # parse maze
    start_position, paths = solve_maze(maze)

    path_lengths = [len(path) for path in paths]
    path_lengths.sort(reverse = True)

    median_index = path_lengths.index(percentile(
        path_lengths, 50, interpolation = 'nearest'))

    print('Median Difficulty: %.4f (index: %d)' % (
        path_lengths[median_index], median_index))
    print('Mean Difficulty: %.4f' % (sum(path_lengths)/
        len(path_lengths)))

    # analysis
    xdata, ydata = array(range(1, len(path_lengths) + 1))
        , array([x for x in path_lengths]) # generate the
        x/y data

    off = 1
    max_x = float(len(path_lengths)) + off*2

    f = lambda x, k, X : (log((-max_x / (x-max_x+off)) -
        1) / (-k)) + X # define curve fitting function
```

```
popt, pcov = curve_fit(f, xdata, ydata, bounds = ([1e
    -10, 1.], [1., max_x])) # fit the curve
perr = sqrt(diag(pcov)) # calculate standard error

print('L: %d' % max_x)
print('popt: %s' % repr(list(popt)))
print('perr: %s' % repr(list(perr)))

plt.plot(xdata, f(xdata, *popt), 'g--')
plt.scatter(xdata, ydata, c = 'r', s = 1, label = '
    data')
plt.xlabel('Index of Path')
plt.ylabel('Path Difficulty')
plt.title('All Paths in Maze')
plt.show()

# data
B = path_lengths[:median_index]
C = path_lengths[median_index:]

print('Ratio of Integrals Score (Sum): %.4f' % (sum(B
    )/sum(C)))
```